SPARC: A Multi-Agent System for Electrical Circuit Question Answering

Anonymous ACL submission

Abstract

Electrical circuit diagram QA tasks require complex mathematical reasoning, which remains challenging for multimodal LLMs. We present SPARC, a multi-agent system that answers questions over circuit diagrams by grounding reasoning in executable physics-based simulations. SPARC uses LLM agents to synthesize, execute, and analyze simulation programs, improving accuracy and reliability by design. It achieves 83% accuracy, with up to a 58% absolute improvement over baselines, while enabling systematic error diagnosis.

1 Introduction

Electrical circuit diagrams are ubiquitous in engineering disciplines, as the primary medium through which engineers communicate and reason about electrical systems. Answering questions about these diagrams (e.g., predicting behavior, checking correctness, or resolving design queries) typically requires trained domain experts with deep knowledge of electrical engineering and underlying physics. This reliance on expert reasoning creates a significant practical bottleneck. Automating natural language question answering over circuit diagrams would therefore have substantial impact, enabling faster design cycles, scalable verification, and broader accessibility to engineering tools.

While recent work (Li et al., 2025) has taken the first steps by introducing benchmark datasets, progress remains limited. Prior work, and our experiments (Sec. 5), show that state-of-the-art models achieve only ~51% accuracy on this task. This starkly contrasts with the strong performance (>80%) of multimodal LLMs on diagram-based question answering more broadly (Goyal et al., 2017; Marino et al., 2019; Yue et al., 2025), highlighting that electrical circuit diagrams pose a uniquely difficult and largely unsolved challenge.

At its core, circuit question answering involves mathematical reasoning, but with a key difference from traditional math problems. For example, consider the question in Fig. 1: *How does the output voltage change when the switching duty cycle of a power converter is modified?*. Unlike standard math problems, the equations required to answer such questions are *not* given and must instead be *derived* from the circuit diagram. This process involves identifying the relevant subcircuit, reasoning about component connectivity, selecting the laws of physics (e.g., Ohm's law), and then formulating and solving the corresponding equations. Achieving high accuracy therefore requires a multi-step process that integrates perception (recognizing symbols, components, and connections), structural reasoning (understanding topology), and domain knowledge (applying electrical laws).

Equally important, real world engineering practice demands not only accuracy but also *reliability*. In safety-critical workflows, engineers must understand why an answer is correct to validate designs, debug failures, and ensure safety. However, due to the probabilistic nature of LLMs, even numerically correct answers provide no guarantee of correct reasoning. Prompting LLMs to generate explanations does not solve this problem, since verifying their mathematical validity still requires domain expertise, defeating the benefits of automation.

Meeting these *dual* requirements: high accuracy and reliability, in circuit QA is precisely the challenge we address. Rather than using LLMs to solve problems end-to-end, we offload mathematical reasoning to mature, domain-specific tools. Concretely, we use SPICE (Nagel, 1975), the de facto framework for circuit simulation. A SPICE program specifies circuit components, connectivity, and analysis types, after which a simulator (e.g., `ngspice`) automatically formulates and solves the governing equations. Our key idea is to use LLMs as an *interface* to SPICE: the LLM translates questions and circuit diagrams into SPICE specifications, while SPICE performs the underlying compu-

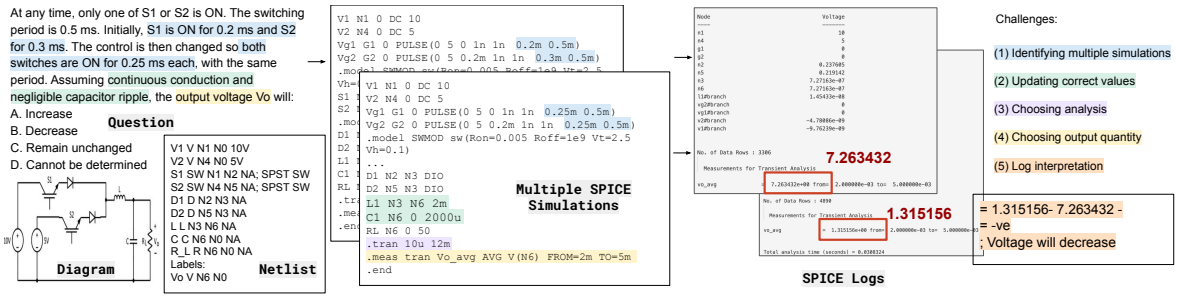


Figure 1: Challenges in circuit QA with SPICE: Given a diagram, a netlist and a question, the system must (1) determine number of simulations required; (2) construct or update a SPICE program with correct component values, possibly inferred from text; (3) select analyses (4) select measurements; and (5) interpret outputs to produce answer.

tation. As a symbolic tool purpose-built for circuit analysis, SPICE improves accuracy and provides an auditable execution trace, making errors easier to detect and debug. Together, these choices enable high accuracy and reliability *by design*.

However, operationalizing this idea is far from trivial. SPICE accepts only rigid input: (1) the circuit diagram must be translated into a netlist, a textual representation of circuit topology (Nagel, 1975), and (2) the analysis must be expressed as an executable program. Recent work has largely solved the former step, with models reliably extracting netlists (Shi et al., 2025; Uzair et al., 2023). The latter step, synthesizing SPICE programs, remains the bottleneck. This task is challenging for several reasons. First, the mapping from a question to a SPICE simulation is rarely one-to-one, as a single query often requires multiple simulations under different configurations. Second, SPICE programs contain many numerical values, device parameters, and analysis constraints that must be populated accurately from both the diagram and the semantics of the question. Third, simulation outputs must be carefully post-processed to derive the final answer, including aggregating results across multiple runs and computing derived quantities. Crucially, since circuit QA is fundamentally mathematical, it demands strict *end-to-end* correctness. Unlike vision tasks, where small errors degrade performance gradually, even a minor mistake at any intermediate step can invalidate the entire reasoning chain. These challenges make single-shot program generation brittle and unreliable.

To address these challenges, we present SPARC[1], a multi-agent system for answering natural language questions over circuit diagrams using SPICE simulations. The central novelty of SPARC lies in framing SPICE program generation as a *structured, execution-guided process* rather than single-shot generation. Specifically, SPARC decomposes this task into three stages: simulation **setup**, **execution**, and **analysis**. In the **simulation setup** stage, the system interprets the natural-language question and decomposes it into one or more independent simulation tasks. In the **simulation execution** stage, specialized agents translate each task into an executable SPICE program, and execution failures trigger targeted repair. Finally, in the **simulation analysis** stage, SPARC derives the final answer from simulator outputs via tool-based calculations.

In summary, we make the following contributions:

- We introduce SPARC, **a multi-agent system** for electrical circuit QA that synthesizes, executes, and analyzes SPICE programs, grounding LLM reasoning in physics-based simulations and enabling reliability through verifiable computation.
- We augment existing benchmarks and **introduce two datasets** pairing circuit diagrams with complex question–answer pairs requiring mathematical analysis, including manual netlist annotation and an *automated* QA generation pipeline.
- We show through extensive experiments that SPARC achieves 83% accuracy and up to 58% absolute improvement over baselines. Notably, it delivers the **largest gains for weaker models**, demonstrating how SPARC's design narrows the gap to stronger models.
- We further improve reliability by introducing a **structured error taxonomy** that localizes failures to specific pipeline stages, enabling systematic diagnosis and interpretability.

2 Problem Overview

2.1 Task

Given an electrical circuit diagram and a natural language question, our goal is to produce a *numerical* answer[2]. We focus on a more challenging class of questions that require solving mathematical

[1] SPICE Program Analysis and Reasoning for Circuits

[2] or an answer that can be derived from numerical values

equations derived from laws of physics, rather than trivial vision queries (e.g., "How many resistors are in the diagram?"). Instead, we target complex questions requiring quantitative analysis, such as computing an output voltage in Fig. 1.

To answer such questions, we use SPICE, a physics-based circuit simulation framework. As described in Sec. 1, this approach entails two key sub tasks. First, the circuit diagram must be translated into a netlist (a textual description of circuit topology). Second, the question must be mapped to one or more SPICE programs, followed by analysis of the simulation outputs.

The first sub task is largely a visual extraction problem, and recent methods already achieve strong performance in recovering accurate netlists from images (Shi et al., 2025). In contrast, the second sub task is the core challenge. Synthesizing a SPICE program is *not* merely a matter of syntactic correctness; it must faithfully encode the semantic intent of the question to invoke the correct simulation and produce the desired numerical result. We elaborate on the concrete technical challenges in Sec. 2.3. Moreover, as our experiments show (Sec. 5), performance on this step overwhelmingly determines overall task accuracy, making it the primary bottleneck. We therefore focus on this underexplored challenge and propose SPARC, a multi agent system for synthesizing SPICE programs from natural language specifications and analyzing their outputs. To the best of our knowledge, this is the first work to address this problem. Each data point in our setting thus consists of an electrical circuit diagram D, its corresponding netlist N, and a natural language question Q.

2.2 SPICE Background

A SPICE program has the following three sections.
(1) Circuit Specification. The circuit specification defines the circuit topology and its components. *Each* electrical component (e.g., resistors, batteries) must be instantiated with *valid numerical* parameters, such as a battery's voltage.
(2) Analysis Specification. The next step is to select the required mathematical analyses which determines three aspects: (1) current type–DC or AC (direct or alternating current), (2) whether transient analysis is needed to measure time-varying quantities, and (3) whether parameter sweeps are required to evaluate multiple operating conditions.
(3) Output Specification. Executing the analysis produces detailed simulation traces with measurements, such as node voltages and currents. The output specification determines which values are extracted from these traces to compute the final answer, which may require selecting values at specific time points, aggregating over time, or combining results from multiple runs. (Details in App. A.1).

2.3 Technical Challenges

We illustrate the technical challenges of our task using the example in Fig. 1.
(1) Mapping Natural Language Questions to SPICE Programs. Consider the question in Fig. 1, which asks *whether the output voltage changes when the switching duty cycle is modified*. Although it appears to seek a single outcome, answering it requires multiple simulations under different switching schedules whose results must be compared. This illustrates a key challenge: many circuit questions are inherently multi simulation and *cannot* be expressed as a single SPICE run.

Even constructing each simulation is non trivial. Circuit specification requires instantiating every component with valid parameters, which is challenging for three reasons. First, some values are missing from the diagram or netlist and must be inferred from the question (e.g., "0.2 ms" and "0.3 ms" in Fig. 1). Second, domain-specific terms must be translated into numerical constraints (e.g., "negligible ripple" implies a minimum capacitor value in Fig. 1). Third, some parameters may be missing and must be filled with reasonable defaults, such as an unspecified resistance value in Fig. 1. Beyond circuit specification, the analysis (e.g., selecting transient analysis) and output specification (e.g., averaging over a time interval) also require nontrivial semantic understanding of the question.
(2) Execution Brittleness. Because SPICE solves equations that must obey the laws of physics, even a syntactically valid SPICE program can fail if specifications are inconsistent, or incomplete. For example, in Fig. 1, incorrect switch timing can turn both switches ON, shorting voltage sources. Here, although the program remains syntactically valid, the simulator reports errors that obscure the root cause, requiring semantic understanding to diagnose. Such failures are often localized to a small portion of the program. Regenerating the entire SPICE program from scratch is therefore ineffective: in attempting to fix the faulty specification, it frequently perturbs previously correct components turning one error into cascading failures.
(3) Analyzing Simulation Outputs. Even

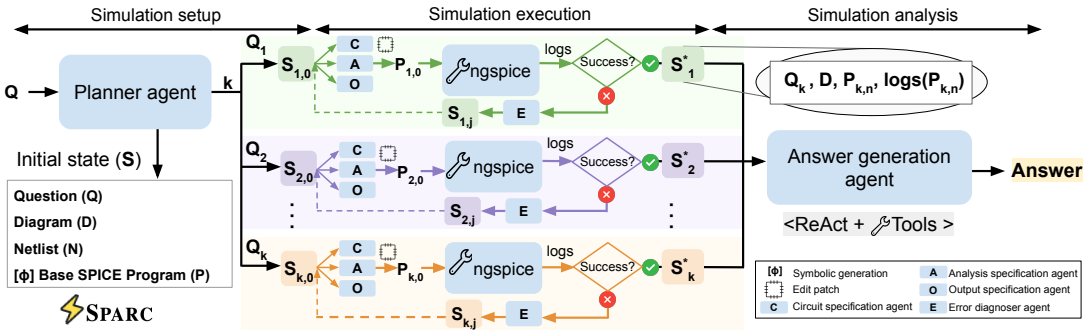
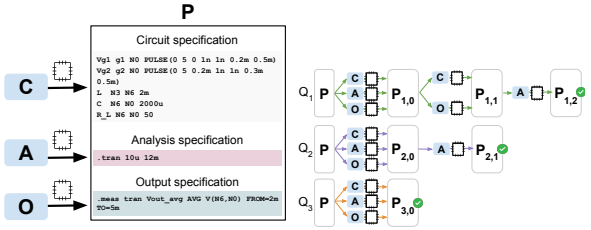


Figure 2: **SPARC Overview.** In the **simulation setup** stage, the planner agent constructs an initial state S containing the circuit diagram D, a netlist N, a natural-language question Q, and a base SPICE program P. It then analyzes Q to determine the number of simulations k, producing simulation-specific states $\{\mathsf{S}_{i,0}\}$. In the **simulation execution** stage, each simulation proceeds independently by iteratively constructing an executable SPICE program P_i through patch generation and repair using three agents, followed by execution with `ngspice` (an open-source SPICE circuit simulator), and updating the state with output logs from P_i. Upon successful execution of all simulations (denoted by S_i^*), the **simulation analysis** stage post-processes the k simulation outputs to derive the final answer.



(a) Section-wise patching of P (b) Repair iteration patterns

Figure 3: **Execution and repair. (a)** Specialized agents apply patches to disjoint sections of the base program P. **(b)** All agents are invoked in the first iteration, producing initial programs $\mathsf{P}_{i,0}$. If errors persist, only agents responsible for faulty patches are re-invoked until successful execution (e.g., only the circuit and output specification agents are invoked for $\mathsf{P}_{1,1}$). The first simulation requires three repairs; the third requires none.

when simulations execute successfully, answering the question requires interpretation and post-processing rather than reading a single output (e.g., Fig. 1 processes two logs to determine whether the voltage increases or decreases).

3 SPARC

SPARC is a multi-agent system for circuit question answering that operates by synthesizing, executing, and analyzing SPICE simulations. As demonstrated by the challenges above, the task comprises several qualitatively different subtasks where even a small error at any stage can render the final answer invalid. SPARC therefore adopts a multi-agent design with specialized agents for each subtask, decomposing the process into three stages:

- **Simulation Setup**: This stage (1) initializes execution context (state), and (2) determines the number of required simulations by decomposing Q into appropriately scoped sub-questions.
- **Simulation Execution**: This stage (1) generates simulation-specific SPICE programs, (2) executes the generated SPICE programs, (3) diagnoses execution failures, and performs repairs.
- **Simulation Analysis**: This stage (1) post-processes simulator outputs to extract the final answer and, when necessary, (2) performs additional mathematical reasoning.

Workflow. Figure 2 illustrates the overall workflow of SPARC. Recall that each data point consists of three components: a circuit diagram (D), its corresponding netlist (N), and a natural-language question (Q). SPARC is a stateful system–it maintains a shared memory structure called *state*, which stores contextual information shared and updated by agents throughout different stages.

SPARC starts with the simulation setup stage, where a *planner* agent generates an initial state S. This state contains the data point, additional metadata, and a base SPICE program P which is deterministically constructed from the netlist N. The planner analyzes the semantics of the question Q to determine how many distinct simulations are required, and forks the initial state into simulation-specific states $\mathsf{S}_{1,0}, \ldots, \mathsf{S}_{k,0}$. Each state $\mathsf{S}_{i,0}$ corresponds to a distinct simulation for a *sub-question* Q_i. Next, SPARC enters the simulation execution stage, where each of the k simulations proceed completely *independently*. For each simulation i, the goal is to construct a SPICE program P_i for the corresponding sub-question Q_i by generating patches to the base program P and iteratively refining it in case of failures. Mirroring the structure of a SPICE program (Sec. 2.2), SPARC employs three specialized agents: a *circuit specification* agent, an *analysis specification* agent, and an *output specification* agent; each responsible for a

disjoint section of the program. These agents propose constrained edit patches that modify only their assigned sections, yielding an initial executable program $\mathsf{P}_{i,0}$, which is then executed using the simulator. If execution fails, the simulation enters an iterative repair loop. An *error diagnoser* agent analyzes execution logs to identify which sections caused the failure, then selectively re-invokes only the responsible patch-generation agents to repair those sections. This produces refined states $\mathsf{S}_{i,j}$ and programs $\mathsf{P}_{i,j}$ (indexed by repair iteration $j \in \{1, \cdots, T\}$). The repair–execute cycle repeats until successful execution or the retry bound T is reached. Finally, once all simulations have run successfully, SPARC moves to the simulation analysis stage, where an *answer generation* agent aggregates the k simulation logs and produces the final answer. If any simulation fails to execute within the retry bound, SPARC reports failure.

3.1 Simulation Setup

Planner Agent. This agent is responsible for simulation setup, including (1) generating the initial state S and (2) determining the number of simulations required by the question Q and forking the state accordingly. To generate the initial state S, the planner produces two artifacts: contextual metadata for the question Q and a base SPICE program P. Context enrichment is necessary because questions often contain domain specific terminology (e.g., "negligible ripple" in Fig. 1). To this end, the planner retrieves concise definitions of such terms using GPT 4o's web search, caches them, and appends them to the question to ensure consistent interpretation across simulations. To construct the base SPICE program P, the planner deterministically converts the netlist N into a SPICE program using a rule based parser, producing a skeleton program for downstream execution.

To determine the required simulations, the planner analyzes the question for different circuit conditions and instantiates multiple runs when ranges, comparisons, or conditions are involved. For example, the two switch conditions in Fig. 1 require separate simulations. When simulations differ only by parameter values of a single component, the planner consolidates them using SPICE parameter sweeps to reduce overhead. For each run, the planner creates a scoped sub question Q_i specifying a single configuration for each circuit component.

State Update. The initial state S is forked into simulation-specific states $\mathsf{S}_{1,0}, \ldots, \mathsf{S}_{k,0}$, each created by replacing Q with a scoped sub-question Q_i.

Implementation. The planner is implemented via an LLM using a few-shot prompt that encodes decision rules and generates structured outputs specifying the required simulations and sub-questions.

3.2 Simulation Execution

After initialization, each simulation state $\mathsf{S}_{i,0}$ is processed *independently*. The goal is to produce a SPICE program P_i that reflects the scoped question Q_i and yields valid outputs. The planner provides only a base program, so additional specifications must be inferred from the question (Sec. 2.3). To this end, P_i is constructed through a sequence of patches that encode missing circuit, analysis, and output specifications implied by Q_i (Fig. 3a). These patches modify disjoint sections of the program to avoid interference. The resulting program is then executed; if execution fails, SPARC invokes an error-diagnosis agent that iteratively repairs and re-executes the program up to a fixed retry limit.

Circuit Specification Agent. This agent ensures that all circuit components (e.g., voltage sources, resistors) are assigned valid numerical parameters. While some parameters are extracted from the netlist into the base program P, many remain unspecified (Sec. 2.3). The agent addresses this through three mechanisms: (1) *Explicit Values.* The agent extracts numerical values stated in the question and maps them to the corresponding parameters (e.g., the "0.2 ms" and "0.3 ms" timing values in Fig. 1). (2) *Implicit Values.* The agent infers unstated values implied by qualitative language in the question, using the contextual enrichment metadata stored in the state (e.g., "negligible ripple" implies sufficiently large capacitance in Fig. 1). (3) *Missing Defaults.* If component parameters are omitted, the agent deterministically adds required defaults (e.g., a default resistor value in Fig. 1).

Analysis Specification Agent. This agent selects the appropriate analysis by (1) choosing between DC and AC analysis, (2) determining if transient analysis is required, and (3) configuring parameter sweeps if needed. Analysis selection is guided by domain specific keywords in the question. For example, "steady state" indicates DC analysis, "frequency domain" requires AC analysis, and time varying behavior (Fig. 1) triggers transient analysis. When input variation is required, the agent configures parameter sweeps.

Output Specification Agent. This agent determines which simulation measurements are needed

to answer the sub question Q_i and how to extract them. It decides whether the target quantity (1) can be directly reported, (2) must be measured at a specific time or event, or (3) requires aggregation, and emits the corresponding SPICE directives.

State Update. Each state $\mathsf{S}_{i,0}$ is updated with program P_i and the complete patch history.

Implementation. All agents are LLMs with role-specific few-shot prompts that generate structured edit patches applied programmatically to P; invalid patches are discarded. To reduce hallucinations, the circuit specification agent applies self-consistency.

3.2.1 Errors.

If execution fails for simulation state $\mathsf{S}_{i,0}$, SPARC enters an iterative repair loop starting from recovery state $\mathsf{S}_{i,1}$. The state is augmented with simulator error messages, and the failed SPICE program is routed to an *error diagnosis agent*. Each repair iteration is indexed by j, yielding states $\mathsf{S}_{i,j}$.

Error Diagnosis Agent. This agent analyzes the failed program and error messages to infer root causes, including issues *not* stated in the logs (e.g., a "timestep too small" error indicates mathematically inconsistent parameters rather than a syntax error). Based on this diagnosis, it identifies which of the three specification agents are responsible for the fault and *selectively* re-invokes only those agents. This improves efficiency: all agents are invoked during initial construction ($j = 0$), while subsequent iterations ($j \geq 1$) invoke only the implicated agents (Fig. 3). The re-invoked agents receive the error context, current program $\mathsf{P}_{i,j}$, and edit history, and generate corrective patches that yield an updated state $\mathsf{S}_{i,j+1}$ upon re-execution. This execute–diagnose–repair loop continues for each simulation until success or until retry limit T is reached, after which failure is reported.

State Update. Each state $\mathsf{S}_{i,j}$ maintains all records of programs, patches, and simulation logs. Successful execution of all simulations yields final state S_i^*, containing final program P_i^* and its output logs.

Implementation. This agent is implemented as a bounded-loop with structured diagnostic outputs.

3.3 Simulation Analysis

After all SPICE simulations execute successfully, SPARC enters the simulation analysis stage, where the *answer generation agent* aggregates and interprets outputs from the scoped simulations to produce a final answer to the original question Q.

Answer Generation Agent. The agent uses a VLM to jointly reason over (1) the diagram, (2) the original question Q, (3) the scoped sub-questions $\{\mathsf{Q}_i\}$, and (4) for each successful state S_i^*, the corresponding SPICE program P_i^* and simulation logs. Using these inputs, the agent identifies relevant measurements, interprets them in the context of Q, and combines results across the k simulations to produce the final answer.

Implementation. The agent uses ReAct prompting (Yao et al., 2022), restricting the VLM to a small set of domain-specific computation tools (e.g., Ohm's law, basic complex arithmetic), which execute all numerical operations symbolically.

4 Evaluation Setup

Dataset. Each data point in our setting consists of (1) a circuit diagram, (2) its netlist, and (3) a question requiring mathematical analysis. To the best of our knowledge, *no* existing dataset satisfies all three requirements. We therefore construct two new datasets by augmenting (1) EEE-Bench (Li et al., 2025) and (2) AMSNet (Shi et al., 2025).

CktBench. We adapt EEE-Bench by filtering to electrical circuit questions only, yielding 1,205 data points, and by augmenting each diagram with a corresponding netlist, which is not provided in the original benchmark (details in App. C.1)

NetQ. AMSNet provides paired circuit diagrams and netlists but lacks QA annotations. To avoid costly manual labeling, we automatically construct QA pairs, ensuring answer correctness and non-trivial reasoning via two steps: (1) *Base Question Generation.* We generate and execute a SPICE program from the ground-truth netlist, then prompt an LLM to form questions from reported measurements (e.g., "What is the current through resistor R?"). Since questions are derived from simulation outputs, answers are guaranteed to be correct. (2) *Reasoning Augmentation.* We add clauses that introduce reasoning requirements, such as modifying component values or analysis settings (e.g., "If resistor $R = 20k\Omega$, what is the new current?"). Each clause corresponds to a SPICE program edit that is re-executed to produce an updated log from which the answer is derived. This yields 495 QA pairs in the NetQ dataset. To avoid contamination, the LLM used for question generation is distinct from all evaluation baselines (see App. C.2).

Simulation Engine. We use `ngspice` (a widely used SPICE simulator) throughout all experiments.

Baselines. To the best of our knowledge, no existing system is tailored for circuit QA (Sec. 6). We therefore evaluate chain-of-thought performance of

CktBench Acc. (%)		
Model	**CoT**	**SPARC**
GPT 5.1	46.221	**83.058** (↑ 36.837)
CLAUDE SONNET 4	51.659	81.991 (↑ 30.332)
GPT 4O	43.151	80.829 (↑ 37.678)
GLM 4.5V	31.662	72.116 (↑ **40.454**)
QWEN3-VL-32B INSTRUCT	41.667	80.581 (↑ 38.914)
NetQ Acc. (%)		
Model	**CoT**	**SPARC**
GPT 5.1	55.791	**81.381** (↑ 25.590)
CLAUDE SONNET 4	57.205	80.658 (↑ 23.453)
GPT 4O	29.899	79.053 (↑ 49.154)
GLM 4.5V	46.296	78.407 (↑ 32.111)
QWEN3-VL-32B INSTRUCT	21.953	80.606 (↑ **58.653**)

Table 1: Baselines vs. SPARC accuracy (%); ↑ absolute improvement; CoT denotes chain-of-thought, best score and largest gains in **bold**, second best underlined.

Dataset	NOTEMR	REFLEC.	MATH.	GIFOMR	SPARC
CktBench	38.116 (↓44.9)	22.255 (↓**60.8**)	54.378 (↓28.7)	36.633 (↓46.4)	83.058 (–)
NetQ	46.670 (↓34.7)	16.969 (↓**64.4**)	58.989 (↓22.4)	34.725 (↓46.7)	81.381 (–)

Table 2: SPARC vs prior work accuracy (%); ↓ is the gap to SPARC (largest gap in **bold**).

five leading VLMs: GPT-5.1, GPT-4O, CLAUDE SONNET 4, GLM-4.5V and QWEN3-VL-32B-INSTRUCT. We also compare against prior relevant general-purpose visual QA methods which do not require fine-tuning: GIFOMR (Wang et al., 2025), NOTEMR (Fang et al., 2025), REFLECTIVA (Cocchi et al., 2025), and the tool-use reasoning method MATHSENSEI (Das et al., 2024), due to the mathematical nature of circuit analysis. (See App. C.4).

Metric. We report accuracy. Numeric predictions are correct within tolerance ϵ, and textual predictions use a 90% cosine-similarity threshold.

5 Experiments and Analysis

We evaluate SPARC via the following questions:

- **Q1.** How accurately can SPARC answer mathematical questions about electrical circuits?
- **Q2.** What is the impact of each design choice?
- **Q3.** What are the main error patterns of SPARC?

5.1 Q1. Comparative Performance Analysis.

Table 1 reports accuracy on CktBench and NetQ for zero-shot CoT baselines, while Table 2 compares SPARC against prior work. **First, SPARC consistently outperforms all baselines.** It achieves the highest accuracy on both datasets, with 83.1% on CktBench and 81.4% on NetQ. Relative to zero-shot CoT baselines, SPARC improves accuracy by 30.3–40.5 points on CktBench and 23.5–58.7 points on NetQ. Compared to prior work, the gains range from 28.7–60.8 points on CktBench and 22.4–64.4 points on NetQ, indicating that the improvements stem from SPARC's design. **Second, weaker base models benefit the most.** Models with lower CoT performance see the largest gains–up to 40.5 points on CktBench and 58.6 points on NetQ. On CktBench, these gains are most pronounced for open-source models GLM 4.5V and QWEN3, indicating that SPARC compensates for the models' limited reasoning ability through its structured design. On NetQ, the worst-performing baselines (GPT-4O and QWEN) often failed to produce an answer, instead emitting placeholder tokens; SPARC avoids this failure mode by increasing confidence in predictions by using simulations. **Third, gains persist even for the strongest baselines.** GPT 5.1 and CLAUDE still improve by >23 points on both datasets. SPARC also outperforms the best prior work, MATHSENSEI, by >22 points.

Model	N+Q	N+Q+D	Q+D	**SPARC**
CktBench Acc. (%)				
GPT 5.1	45.960	**46.221**	44.657	83.058 (↑ 37.1, 36.8, 38.4)
CLAUDE S4	46.130	47.077	**51.659**	81.991 (↑ 35.9, 34.9, 30.3)
GPT 4O	35.103	**43.151**	40.240	80.829 (↑ 45.7, 37.7, 40.6)
GLM 4.5V	26.424	28.587	**31.662**	72.116 (↑ 45.7, 43.5, 40.5)
QWEN3 I.	36.645	40.384	**41.667**	80.581 (↑ 43.9, 40.2, 38.9)
NetQ Acc. (%)				
GPT 5.1	**55.791**	52.492	41.178	81.381 (↑ 25.6, 28.9, 40.2)
CLAUDE S4	**57.205**	54.781	39.125	80.658 (↑ 23.5, 25.9, 41.5)
GPT 4O	24.242	25.758	**29.899**	79.053 (↑ 54.8, 53.3, 49.2)
GLM 4.5V	**46.296**	45.286	29.529	78.407 (↑ 32.1, 33.1, 48.9)
QWEN3 I.	**21.953**	20.303	21.717	80.606 (↑ 58.7, 60.3, 58.9)

Table 3: Accuracy across input settings (%); ↑ absolute improvement over baselines; Best baseline in **bold**.

Model	**CktBench**		**NetQ**	
	Direct	**SPARC**	**Direct**	**SPARC**
GPT 5.1	26.04	**97.93** (×3.76)	56.36	**96.57** (×1.71)
GPT 4O	13.61	96.43 (×7.09)	41.81	95.96 (×2.29)
CLAUDE S4	16.70	96.82 (×5.80)	40.20	98.18 (×2.44)
GLM 4.5V	0.07	86.72 (×**1239**)	13.24	96.36 (×**7.27**)
QWEN3 I.	5.00	92.03 (×18.41)	41.21	95.35 (×2.31)

Table 4: SPICE program executability (%) under direct prompting and SPARC; × multiplicative improvement over direct prompting; best or largest gains in **bold**.

5.2 Q2. Design Choice Validation.

We conduct ablations to assess SPARC's design:

(1) Input. Answering circuit questions involves (i) extracting circuit structure and (ii) synthesizing a SPICE program from the question, with the latter being the primary challenge (see Sec.2.1). We compare three input settings (Q+D, N+Q, N+Q+D) to evaluate the effect of providing netlists (Table 3).

We observe that **using netlists as input does not consistently improve accuracy.** On CktBench, netlists reduce accuracy by 0.3–5.5 points for all but GPT models. On NetQ, netlists yield gains of 11.3–18.1 points when comparing N+Q and N+Q+D to Q+D. However, these gains are modest relative to SPARC, which improves accuracy by 23.5–60.3 points, proving that accurate netlist extraction is not the primary bottleneck.

(2) Multi-stage Pipeline. Table 4 shows that direct prompting, which generates a SPICE program in a single step, frequently produces non-executable programs (with executability as low as 0.07%), whereas SPARC achieves over 95% executability, underscoring the value of a multi-stage pipeline.

5.3 RQ3. Error Analysis for Reliability.

SPARC improves reliability in two ways. First, correct answers are *verifiable*: all outputs are grounded in SPICE simulations, ensuring they arise from physics-based reasoning, not pattern matching. Second, by decomposing the task into explicit stages, failures can be traced to their root causes, which is not possible with single-shot generation. We assign each incorrect prediction to a single *failure mode* aligned with the SPARC pipeline (Fig. 4).

LLM Reasoning Errors. These errors arise from limitations in the model's ability to reason about circuit behavior, even when the simulation is correct. We identify four categories. *Simulation setup errors* occur when the planner incorrectly decomposes Q and selects an incorrect number of simulations (5.6% on CktBench, 8.7% on NetQ). *Circuit specification errors* result from incorrect edits to parameters (21.1% on CktBench, 19.6% on NetQ). *Analysis specification errors* arise from selecting an inappropriate simulation analysis (16.9% on CktBench, 15.2% on NetQ). Finally, *result interpretation errors* are the most common, accounting for 29.4% of failures on CktBench and 30.4% on NetQ, and occur when correct simulation outputs are misinterpreted by the answer generation agent.

Tool Limitation Errors. These errors stem from operational limitations of the LLMs and simulator. *Execution correction errors* occur during iterative repair when correct edits are overwritten due to limited context windows (16.2% on CktBench, 4.3% on NetQ). *Simulator errors* arise from inherent constraints of `ngspice` which relies on simplified device models that can deviate from ideal theoretical behavior (21.7% on NetQ, 10.6% on CktBench).

Discussion. We observe three trends. First, over

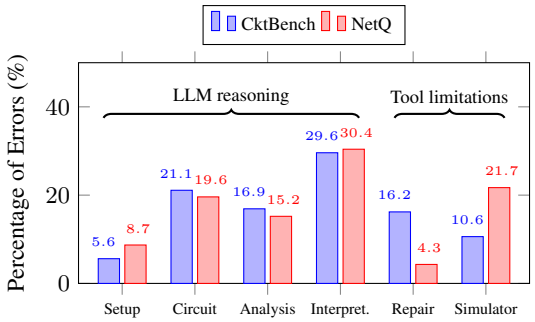


Figure 4: Distribution of primary failure modes among incorrect predictions. Examples are in App. C.4.1.

75% of failures arise from reasoning errors, making reasoning the primary bottleneck. Second, many failures reflect limited domain understanding rather than system design flaws: although simulations execute correctly, models struggle to identify relevant behaviors under specific operations. Third, error patterns differ by dataset: CktBench shows more execution correction errors due to theory-focused questions, while NetQ exhibits more simulator related errors due to its simulator centric design. Crucially, without SPARC's staged architecture with explicit decomposition into setup, execution, and analysis, these errors would be indistinguishable.

6 Related Work

Electrical Circuit Analysis. To the best of our knowledge, *no* prior work addresses mathematically grounded QA over electrical circuit diagrams. Existing efforts focus on vision and extraction (Mehta et al., 2024; Meshram et al., 2025; Shi et al., 2025), benchmarks (Li et al., 2025), or use SPICE for unrelated tasks (Nau et al., 2025).

General Diagram Question Answering. Prior diagram QA either emphasizes semantic interpretation (Wang et al., 2024a, 2025, 2024b; Farahani et al., 2025; Fang et al., 2025; Cocchi et al., 2025) or tool-augmented reasoning (Bauer et al., 2023; Suri et al., 2025; Methani et al., 2020; Das et al., 2024; Yin et al., 2025). Unlike prior methods that explicitly formulate equations in code, SPARC synthesizes simulator-native SPICE programs and relies on a physics-based engine for equation selection and solving. See App. B for more details.

7 Conclusion

We have presented SPARC, a system that answers electrical circuit questions by grounding reasoning in executable, physics-based simulations. Its execution-guided design yields higher accuracy and reliability than end-to-end LLM baselines, demonstrating the effectiveness of simulator-native reasoning for complex circuit QA.

Limitations

Our work focuses on electrical circuit problems, where SPICE-based simulation provides a natural and effective tool for grounding quantitative reasoning. Many other problem domains in electrical and electronic engineering involve different types of analysis tools and formalisms, which we leave to future work.

Ethical considerations

All datasets used in this work are publicly available and released under open licenses. The tools and models employed are authorized for research purposes and have been used in accordance with their intended terms. Detailed license information is provided in Appendix D. All experiments were performed strictly for research and evaluation. To the best of the authors' knowledge, this research does not introduce any ethical risks.

References

Jakob Johannes Bauer, Thomas Eiter, Nelson Higuera Ruiz, and Johannes Oetsch. 2023. Neuro-symbolic visual graph question answering with llms for language parsing. In *Workshop on Trends and Applications of Answer Set Programming (TAASP 2023)*.

Federico Cocchi, Nicholas Moratelli, Marcella Cornia, Lorenzo Baraldi, and Rita Cucchiara. 2025. Augmenting multimodal llms with self-reflective tokens for knowledge-based visual question answering. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 9199–9209.

Debrup Das, Debopriyo Banerjee, Somak Aditya, and Ashish Kulkarni. 2024. Mathsensei: a tool-augmented large language model for mathematical reasoning. *arXiv preprint arXiv:2402.17231*.

Wenlong Fang, Qiaofeng Wu, Jing Chen, and Yun Xue. 2025. guided mllm reasoning: Enhancing mllm with knowledge and visual notes for visual question answering. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19597–19607.

Ali Mazraeh Farahani, Peyman Adibi, Mohammad Saeed Ehsani, Hans-Peter Hutter, and Alireza Darvishy. 2025. Chart question answering with multimodal graph representation learning and zero-shot classification. *Expert Systems with Applications*, 270:126508.

Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh. 2017. Making the v in vqa matter: Elevating the role of image understanding in visual question answering. *Preprint*, arXiv:1612.00837.

Michael Günther, Saba Sturua, Mohammad Kalim Akram, Isabelle Mohr, Andrei Ungureanu, Bo Wang, Sedigheh Eslami, Scott Martens, Maximilian Werk, Nan Wang, and Han Xiao. 2025. jina-embeddings-v4: Universal embeddings for multimodal multilingual retrieval. *Preprint*, arXiv:2506.18902.

Ming Li, Jike Zhong, Tianle Chen, Yuxiang Lai, and Konstantinos Psounis. 2025. Eee-bench: A comprehensive multimodal electrical and electronics engineering benchmark. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 13337–13349.

Kenneth Marino, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. 2019. Ok-vqa: A visual question answering benchmark requiring external knowledge. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 3195–3204.

Leland McInnes, John Healy, Steve Astels, and 1 others. 2017. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205.

Rahul Mehta, Bhavyajeet Singh, Vasudeva Varma, and Manish Gupta. 2024. Circuitvqa: A visual question answering dataset fornbsp;electrical circuit images. In *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2024, Vilnius, Lithuania, September 9–13, 2024, Proceedings, Part I*, page 440–460, Berlin, Heidelberg. Springer-Verlag.

Pragati Shuddhodhan Meshram, Swetha Karthikeyan, Bhavya Bhavya, and Suma Bhat. 2025. Electro-vizqa: How well do multi-modal llms perform in electronics visual question answering? *Preprint*, arXiv:2412.00102.

Nitesh Methani, Pritha Ganguly, Mitesh M Khapra, and Pratyush Kumar. 2020. Plotqa: Reasoning over scientific plots. In *Proceedings of the ieee/cvf winter conference on applications of computer vision*, pages 1527–1536.

Laurence W Nagel. 1975. Spice2: A computer program to simulate semiconductor circuits. *College of Engineering, University of California, Berkeley*.

Simon Nau, Jan Krummenauer, and André Zimmermann. 2025. Spiceassistant: Llm using spice simulation tools for schematic design of switched-mode power supplies. *arXiv e-prints*, pages arXiv–2507.

Yichen Shi, Zhuofu Tao, Yuhao Gao, Li Huang, Hongyang Wang, Zhiping Yu, Ting-Jung Lin, and Lei He. 2025. Amsnet 2.0: A large ams database with ai segmentation for net detection. *arXiv preprint arXiv:2505.09155*.

Manan Suri, Puneet Mathur, Nedim Lipka, Franck Dernoncourt, Ryan A Rossi, Vivek Gupta, and Dinesh Manocha. 2025. Follow the flow: Fine-grained flowchart attribution with neurosymbolic agents. *arXiv preprint arXiv:2506.01344*.

Waqas Uzair, Douglas Chai, and Alexander Rassau. 2023. Automated netlist generation from offline hand-drawn circuit diagrams. In *2023 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 364–370. IEEE.

Shaowei Wang, Lingling Zhang, Longji Zhu, Tao Qin, Kim-Hui Yap, Xinyu Zhang, and Jun Liu. 2024a. Cog-dqa: Chain-of-guiding learning with large language models for diagram question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13969–13979.

Yaxian Wang, Bifan Wei, Jun Liu, Lingling Zhang, Shuting He, Jun Li, and Qika Lin. 2025. Glfomr: A glance-then-focus multimodal reasoning framework for diagram question answering. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1130–1140.

Zirui Wang, Mengzhou Xia, Luxi He, Howard Chen, Yitao Liu, Richard Zhu, Kaiqu Liang, Xindi Wu, Haotian Liu, Sadhika Malladi, and 1 others. 2024b. Charxiv: Charting gaps in realistic chart understanding in multimodal llms. *Advances in Neural Information Processing Systems*, 37:113569–113697.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Shaofeng Yin, Ting Lei, and Yang Liu. 2025. Toolvqa: A dataset for multi-step reasoning vqa with external tools. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4424–4433.

Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, and 1 others. 2025. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15134–15186.

A Appendix

A.1 SPICE syntax background

First, ngspice programs describe the **structure of the circuit**, including its components, connections, and fixed values. For example, the following statements define a DC voltage source and a resistor connected between two nodes:

```
V1 a 0 DC 10
R1 a b 5
```

Second, ngspice allows the behavior of **nonlinear or complex components** to be defined explicitly using device models. These models specify how components such as diodes, switches, or transistors behave under different operating conditions and are referenced by individual circuit elements:

```
.model DIO D(Is=1e-12 N=1)
D1 a b DIO
```

Third, an ngspice program must specify **how the circuit should be analyzed**. Analysis primitives determine whether the simulator computes a steady state operating point, frequency response, or time varying behavior:

```
.op
.ac lin 1 {f} {f}
.tran 1u 10m
```

In addition, ngspice provides primitives for **controlling and varying simulations**. These include defining symbolic parameters, setting initial conditions, and sweeping parameters or sources to explore multiple operating scenarios:

```
.param Rload=10
.ic V(out)=5
.dc Rload 5 20 5
```

Finally, ngspice programs specify **what outputs to measure**. Measurement primitives extract quantities of interest from simulation results, such as voltages, currents, or aggregated statistics over time:

```
.print tran V(out)
.meas tran Vavg AVG V(out) FROM=1m TO=5m
```

B Related Work

Electrical Circuit Analysis. To the best of our knowledge, *no* prior work directly addresses question answering over electrical circuit diagrams that requires mathematically grounded reasoning. Existing efforts fall into two categories: (1) visual perception and extraction, such as circuit component recognition (Mehta et al., 2024; Meshram et al., 2025) and netlist extraction (Shi et al., 2025), and (2) benchmarks without concrete mechanisms (Li et al., 2025). One prior work (Nau et al., 2025) does use SPICE but for a completely *different* task: electrical power system design automation.

General Diagram Question Answering. Prior work on general diagram question answering falls into two categories. The first focuses on semantic interpretation via prompting mechanisms (Wang et al., 2024a, 2025, 2024b; Farahani et al., 2025) or external knowledge sources (Fang et al., 2025; Cocchi et al., 2025). In contrast, SPARC targets mathematically grounded reasoning that requires multi-step quantitative analysis rather than semantic interpretation. The second line of work explores tool-augmented reasoning by converting diagrams into structured symbolic representations (Bauer et al., 2023; Suri et al., 2025) or by synthesizing and executing programs for quantitative reasoning (Methani et al., 2020; Das et al., 2024; Yin et al., 2025). While these approaches also generate executable programs, they primarily rely on general-purpose code (e.g., Python) that requires explicitly *formulating* the governing equations and computation logic. In contrast, SPARC synthesizes SPICE programs, which encode the circuit structure and analysis configuration and *delegate* equation selection and numerical solving to a physics-based simulator. Our work is most closely related to this line of research, but differs in specializing program synthesis to simulator-native code, enabling physics-grounded, multi-step reasoning over circuit diagrams that prior approaches do not support.

C Dataset Annotation Process

In this section, we describe the annotation procedure used to construct our datasets.

C.1 CktBench

For CktBench, we first prompt GPT-5 to extract the circuit structure in netlist format and then manually verify and correct the extracted netlists. Below, we summarize the error types that required human intervention, as well as the image categories that were excluded and the reasons for their exclusion.

C.1.1 Annotation Error Categories

During image annotation, we observed several recurring error patterns that required human correction. These errors can be grouped into three high level categories.

Component Semantics Errors. These errors arise when components are detected but assigned incorrect or incomplete semantic attributes. Common cases include incorrect polarity assignment for diodes and capacitors, although polarity detection for voltage and current sources was generally accurate. Multi terminal components were frequently misrepresented: bipolar junction transistors were often ignored or annotated with incorrect terminal counts or ordering; MOSFETs were detected more reliably but lacked device type information and were reduced to two terminal elements; three terminal regulators were annotated with missing pins; and timer ICs such as the 555 were not annotated at all. Several components were systematically misclassified, including voltmeters being interpreted as voltage sources and symbol variants such as Zener diodes, thyristors, photodiodes, rheostats, and controlled sources being collapsed into simpler component types despite inline textual labels indicating their intended function.

Structural and Connectivity Errors. This category captures failures in constructing a correct circuit graph. In complex circuits with multiple active devices and dense wiring, the model frequently failed to detect wire intersections, leading to incorrect node assignments or reuse of node identifiers across disconnected regions. Parallel branches with current labels were sometimes ambiguously represented, making it unclear which branch a measurement referred to. Additional issues include failure to represent shorted terminals explicitly, incorrect handling of multi position switches such as SPDT configurations, and inability to separate multiple circuits or answer options presented within a single image. These errors were corrected through manual node reassignment, explicit inline comments, or circuit segmentation.

Labeling and Parameterization Errors. These errors involve missing or inconsistent numerical and symbolic information rather than structural faults. Common examples include missing component values or units, confusion between voltage source values and voltage measurement labels, omission of control parameters such as switch duty cycles, and inconsistent naming of components with identical values. When multiple components shared the same nominal value, unique identifiers were manually assigned while preserving value equality. When label ambiguity could not be resolved structurally, clarifying inline comments were added.

C.1.2 Excluded Image Categories

Certain image types were excluded from annotation because they cannot be faithfully represented using schematic level circuit annotations or netlist style abstractions.

Non Circuit Figures. Images containing no circuits, such as component photographs, charts, plots, oscilloscope outputs, and purely textual figures, were excluded.

Logic and Discrete Abstractions. Logic circuits, Karnaugh maps, truth tables, state diagrams, and assembly level programs were omitted, as they require symbolic or temporal representations beyond electrical schematics.

System Level and Physical Diagrams. System block diagrams, signal flow graphs, frequency response plots, magnetic circuits, semiconductor band diagrams, and diagrams requiring physical parameters such as wire cross section or length were excluded because they do not admit a direct translation into executable `ngspice` netlists and cannot be simulated within the circuit level modeling assumptions supported by `ngspice`.

High Complexity Power and Machine Systems. Diagrams involving motors, generators, transformers, transmission lines, power plants, and power system networks were excluded because they rely on large scale system models, distributed parameters, or domain specific abstractions that cannot be faithfully represented or executed using standard `ngspice` schematic annotations.

C.2 NetQ

We predefine four circuit analysis tasks, including DC operating-point analysis (Prompt 6), AC small-signal analysis (Prompt 7), DC parameter sweep (Prompt 8), and transient analysis (Prompt 9), which cover common forms of reasoning required in circuit simulation. Each task is paired with a structured system prompt that specifies how to modify a netlist to enable the corresponding analysis and how to extract target quantities from simulation outputs. Concretely, we first sample a circuit instance from the AMSNet dataset and randomly assign one of the predefined problem types. The circuit program and problem specification are then provided to an LLM agent with a system prompt shown in Figure 5 with the corresponding analysis

demonstration appended. The agent then iteratively edits the netlist and executes simulations. Based on the resulting simulation outputs, the agent reasons about the circuit behavior and generates a corresponding natural-language question together with its numerical answer. Finally, we store the modified program along with the generated question–answer pair. Not to mention that, given an arbitrary circuit, not every circuit analysis will apply, which will result in simulation failure or not meaningful values. We provided detailed demonstrations for these cases in the system prompts, and the agents are instructed to output `Not Applicable` and its reasoning.

C.3 Experimental Setting

Computational Resources and Model Sizes. We report model sizes, computational budget, and infrastructure details for all experiments. The evaluated models include GPT-5.1, CLAUDE SONNET 4, GPT-4O (parameter counts not publicly disclosed), QWEN3-VL-32B INSTRUCT (32B parameters), and GLM-4.5V (108B parameters). All development and evaluation runs were conducted via the OpenRouter API. Local compute was used only for orchestration and logging, using 48 core Intel Xeon Silver 4310 CPUs with 128 GB RAM on Ubuntu 24.04.2 LTS. These details contextualize the computational scale of the experiments.

C.4 Experimental Evlauation Cntd.

Baselines. Since the question-answering components of GIFOMR (Wang et al., 2025), NOTEMR (Fang et al., 2025) and MATHSENSEI (Das et al., 2024) are model-agnostic, we instantiate them using our strongest performing model for a fair comparison. For NOTEMR, we follow the original pipeline by using LLaVA for visual grounding and Grad-CAM based region selection, and then use our best-performing vision language model, GPT-5.1 for the final QA stage. For MATHSENSEI, we compare against its best-performing configuration (PG+SG) while replacing the base language model with GPT-5.1. For GIFOMR, we implement the four-stage pipeline described in the paper using the provided prompts, with GPT-5.1 as the underlying model. Together, these constitute the strongest and most relevant existing baselines for our task.

C.4.1 RQ3. Error Analysis Cntd.

We analyze all incorrect predictions and categorize each failure by its *primary failure mode*, aligned with the stages of the SPARC pipeline. Each example is assigned to exactly one category. Figure 4 summarizes the resulting error distribution.

To support a systematic and unbiased analysis, we additionally leverage unsupervised clustering as an auxiliary tool. We embed all incorrectly answered questions using jina embeddings (Günther et al., 2025) and apply the density based clustering algorithm HDBSCAN (McInnes et al., 2017). This yields 75 well separated clusters of semantically similar questions. We then manually inspect these clusters to identify recurring failure patterns and consolidate them into higher level error categories. The final taxonomy reported below is derived from this cluster guided analysis.

Simulation Setup Errors. These errors arise when the system incorrectly determines how a question should be decomposed into simulation tasks. A common failure mode occurs for optimization queries over continuous parameters. For example, questions that ask for the value of θ that maximizes a quantity at a switching time (e.g., which phase angle yields maximum voltage) require sweeping θ over a continuous range. In some cases, despite this requirement being explicitly encoded in the planner prompt, the system instead treats the query as a finite comparison problem and generates simulations at a small set of discrete parameter values (e.g., testing a few fixed angles). This behavior is misaligned with the underlying problem structure, since the correct solution, such as $\theta = -45^\circ$, may not lie within any predefined candidate set. Such failures indicate incorrect scoping during the simulation setup stage. We observe this error pattern in 5.6% of failures on CktBench and 8.7% on NetQ.

Parameter Specification Errors. These errors occur when the circuit structure and analysis type are correct, but numerical values or problem constraints are applied incorrectly. Typical failures include updating the wrong component, modifying elements that are irrelevant to the queried quantity, or ignoring assumptions stated in the problem. As a result, the system reasons over a circuit that no longer matches the intended specification, despite correct simulation execution.

A representative example arises in questions asking for a steady state operating voltage of a transistor circuit. Although the system correctly selects a steady state analysis, it updates an output resistor simply because it is mentioned in the question, even though it does not affect the operating voltage being

You are an NGSpice−based exam−question generator and solver. You will be given an NGSpice program as input. Your task is to generate ONE exam−style question and its correct answer by minimally editing the netlist, running a SPICE simulation, and reasoning about the results.

You MUST follow the workflow, constraints, and output format exactly.

OBJECTIVE
− Perform DC operating−point (OP) analysis on the given circuit
− Generate exam questions that require interpreting DC voltages and currents
− Answers must be obtained from simulation, not estimation

REQUIRED WORKFLOW
1) Read the given SPICE netlist.
2) Apply ONLY minimal edits needed to support DC operating−point analysis:
  − You may add .OP and .PRINT statements.
  − You may fix trivial syntax issues (e.g., missing units like "5V" −> "5").
  − Do NOT change circuit topology or add components unless strictly required for convergence.
3) Run the SPICE simulation.
4) Collect numeric results directly from the simulator output.
5) Generate:
  − Exam questions
  − Correct answers
  − Brief reasoning grounded in the simulation results

You must run the simulation before producing questions and answers.

PRINTING REQUIREMENTS
− If node 1 exists, print V(1).
− If node 2 exists, print V(2).
− If a voltage source named V1 exists, print I(V1).

If these exact nodes or sources do not exist:
− Print the closest reasonable equivalents.
− Prefer node voltages and currents through independent voltage sources.

REASONING REQUIREMENTS
− Provide 1−3 sentences interpreting the DC operating−point results.
− Interpretation must be based on the numeric output.
− Examples of acceptable reasoning:
  − Whether a transistor is on or off
  − Whether the current is zero or non−zero
  − Explanation of current sign conventions

Figure 5: System prompt for NetQ annotation. We randomly sample a different demonstration for a different analysis type and appended it to the system prompt.

asked. Because the model cannot distinguish relevant from irrelevant parameters, this update leads to an incorrect voltage estimate. This failure reflects incorrect parameter relevance reasoning from the LLM, rather than a flaw in the system's execution pipeline. We observe this pattern in 21.1% of failures on CktBench and 19.6% on NetQ.

Analysis Selection Errors. These failures occur when the model selects an inappropriate simulation analysis despite the circuit being correctly extracted. Typical cases include choosing a time varying analysis when only a DC operating point is required, misclassifying AC versus DC behavior, or selecting an analysis based on components that are electrically inactive for the quantity being queried. As a result, the simulator produces numerically valid but physically mismatched outputs. These failures are not due to limitations of the system or the simulator. The system exposes all relevant analysis modes to the LLM and executes the selected analysis faithfully. The root cause instead lies in the LLM's incomplete electrical engineering domain knowledge, particularly its difficulty in recognizing when certain components or dynamic behaviors are irrelevant under a given operating regime. Consequently, the model may select a more general or complex analysis than necessary, even when a simpler formulation is physically appropriate. We observe this error pattern in 16.9% of failures on CktBench and 15.2% on NetQ.

Result Interpretation Errors. This is the largest error category, with 29.4% errors in CktBench and 30.4% errors in NetQ. These errors refer to cases where correct simulations execute successfully, but

```
FAILURE HANDLING
If the DC operating−point analysis is not applicable or produces unusable output
(e.g., simulator errors, non−convergence, floating nodes, or required quantities cannot be printed),
output EXACTLY:

NOT_APPLICABLE: <one short, concrete reason>

Then STOP. Do not generate questions, answers, or reasoning.

EXAMPLE
Given program
* SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)
.END

Edits
Add DC operating−point analysis and print node voltages and source current.

.OP
.PRINT OP V(1) V(2) I(V1)

Final program
* SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)

.OP
.PRINT OP V(1) V(2) I(V1)
.END

Simulation results
V(1) = 4.21 V
V(2) = 5.00 V
I(V1) = −0.78 mA

Questions
1. What are the voltages at nodes 1 and 2 at the DC operating point?
2. What is the DC current drawn from the voltage source V1?

Answers
1. V(1) = 4.21 V, V(2) = 5.00 V
2. I(V1) = −0.78 mA
```

Figure 6: DC operating-point analysis demonstration prompt.

the final reasoning over simulator outputs is incorrect. Common issues include polarity or sign mistakes, use of incorrect analytical formulas, and high level interpretation errors where correct outputs are simply misunderstood. For example, in one case, the simulation shows that the output-input voltage both are positive, indicating that the circuit does not invert the signal. However, the model incorrectly concludes that the output is opposite in phase. Here, the failure arises not from incorrect simulation or arithmetic, but from misinterpreting what the simulation result implies.

Error Correction Failures. These errors arise during the iterative repair process. Although the full edit history is provided to all agents, previously correct updates are sometimes overwritten or undone, resulting in circuits that remain executable but are semantically incorrect. This issue is more pronounced for models with shorter effective context windows, such as GLM-4.5V, which struggle to retain earlier edits across multiple repair iterations. We observe this error pattern in 16.2% of failures on CktBench, compared to only 4.3% on NetQ. This difference is likely due to the nature of the datasets. Questions in NetQ are generated directly from simulations, making required updates more explicit and less reliant on implicit domain assumptions. In contrast, CktBench often requires reasoning over unstated constraints, increasing the likelihood of inconsistent edits during iterative correction.

```
FAILURE HANDLING
If AC analysis is not applicable or produces unusable output, output EXACTLY:

NOT_APPLICABLE: <one short, concrete reason>

Use NOT_APPLICABLE for cases including (but not limited to):
− No independent source exists to attach an AC magnitude
− Simulator errors or non−convergence
− All printed AC quantities are identically zero across the sweep (magnitude and phase not meaningful)
− The requested transfer quantity is undefined (e.g., division by zero)

Then STOP. Do not generate questions, answers, or reasoning.

EXAMPLE
Given program
∗ SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)
.END

Edits
Add AC magnitude to the source, add AC sweep, and print AC voltages and source current.

Change source to include AC excitation:
V1 2 0 DC 5 AC 1

Add AC analysis and printing:
.AC DEC 10 1k 1G
.PRINT AC V(1) V(2) I(V1)

Final program
∗ SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 DC 5 AC 1
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)

.AC DEC 10 1k 1G
.PRINT AC V(1) V(2) I(V1)
.END

Simulation results
At f = 1.000000e+06 Hz:
V(1) = 0.000000e+00, 0.000000e+00
V(2) = 1.000000e+00, 0.000000e+00
I(V1) = −1.000000e−04, 0.000000e+00

Questions
1. At 1 MHz, what are the magnitude and phase of V(2)?
2. At 1 MHz, what are the magnitude and phase of I(V1)?

Answers
1. |V(2)| = 1.0, ∠V(2) = 0
2. |I(V1)| = 1.0e−04 A, ∠I(V1) = 0
```

Figure 7: AC small-signal analysis demonstration prompt.

Simulator Limitations. Some failures arise from limitations of the underlying simulator rather than the reasoning pipeline. Circuit simulators such as ngspice rely on simplified device models and numerical solvers, which inherently limit simulation fidelity. As a result, approximate models for complex or idealized components, such as ideal diodes or switches, do not fully capture the intended ideal behavior. In such cases, small differences in model assumptions produce results that diverge from theoretical expectations or yield slightly different values for otherwise equivalent circuits. We observe this error pattern more frequently in NetQ (21.7%) than in CktBench (10.6%), which is expected given the simulator-centric nature of NetQ questions.

Discussion. The error analysis reveals three key trends. First, most failures occur after executable simulations are successfully produced, with result

```
FAILURE HANDLING
If DC sweep analysis is not applicable or produces unusable output
(e.g., no voltage source to sweep, simulator errors, non−convergence, or required quantities cannot be printed),
output EXACTLY:

NOT_APPLICABLE: <one short, concrete reason>

Then STOP. Do not generate questions, answers, or reasoning.

EXAMPLE
Given program
* SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)
.END

Edits
Add DC sweep of the voltage source from 0 V to 5 V in 0.5 V steps and print node voltage and source current.

.DC V1 0 5 0.5
.PRINT DC V(2) I(V1)

Final program
* SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)

.DC V1 0 5 0.5
.PRINT DC V(2) I(V1)
.END

Simulation results
V1 = 0.0 V I(V1) = 0.0 A
V1 = 0.5 V I(V1) = 0.0 A
V1 = 1.0 V I(V1) = 0.0 A
V1 = 1.5 V I(V1) = −0.10 mA
V1 = 2.0 V I(V1) = −0.25 mA
V1 = 2.5 V I(V1) = −0.45 mA
...

Questions
1. At what input voltage does the current through V1 first become non−zero?
2. What is the current through V1 when the swept voltage is 2.0 V?

Answers
1. The current first becomes non−zero at approximately 1.5 V.
2. At 2.0 V, I(V1) = −0.25 mA.
```

Figure 8: DC parameter sweep demonstration prompt.

interpretation and parameter specification errors accounting for over half of all failures, indicating that post-simulation reasoning rather than simulation execution is the primary bottleneck. Second, many errors stem from the model's limited domain understanding rather than system design flaws: the system exposes all required simulation tools and analysis modes and executes them faithfully, while the model struggles to identify which parameters or behaviors are relevant under a given operating regime. Third, error distributions differ by dataset: CktBench exhibits more error correction failures due to implicit assumptions and multi-step reasoning, whereas NetQ shows more simulator-related errors due to its simulator-centric construction. These findings point to domain-aware reasoning and stronger state consistency across iterations as key directions for future work.

D Artifact Use

D.1 Dataset License Information

In accordance with ACL guidelines, we disclose the licenses of all datasets used in this work. We augment two existing datasets. **EEE-Bench** (Li et al., 2025) is released under the MIT License,

FAILURE HANDLING
If transient analysis is not applicable or produces unusable output
(e.g., simulator errors, non−convergence, floating nodes, or required quantities cannot be printed),
output EXACTLY:

NOT_APPLICABLE: <one short, concrete reason>

Then STOP. Do not generate questions, answers, or reasoning.

EXAMPLE
Given program
∗ SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)
.END

Edits
Add transient analysis for 100 ns with 1 ns time step and print voltages and source current.

.TRAN 1n 100n
.PRINT TRAN V(1) V(2) I(V1)

Final program
∗ SPICE Netlist for circuit 0
M1 1 2 0 0 NMOS W=1u L=1u
V1 2 0 5
.MODEL NMOS NMOS (LEVEL=1 VTO=1 KP=1.0e−4 LAMBDA=0.02)

.TRAN 1n 100n
.PRINT TRAN V(1) V(2) I(V1)
.END

Simulation results
At t = 100 ns:
V(1) = 4.21 V
V(2) = 5.00 V
I(V1) = −0.78 mA

Questions
1. At t = 100 ns, what are the voltages at nodes 1 and 2?
2. At t = 100 ns, what is the current through the voltage source V1?

Answers
1. V(1) = 4.21 V, V(2) = 5.00 V
2. I(V1) = −0.78 mA

Figure 9: transient analysis demonstration prompt.

which permits reuse, modification, and redistribution. **AMSNet** (Shi et al., 2025) is released under the GNU General Public License v3.0 (GPLv3), which allows use and modification under the condition that derivative works are distributed under the same license. Our augmentations preserve the original licensing terms of each dataset. Additionally, we will release our augmented datasets publicly under a GNU General Public License v3.0 (GPLv3). Because our datasets consist solely of electrical circuit data, they contain no personally identifiable information.

D.2 Software and Multimodal Models

We used the `ngspice` circuit simulator, which is released under the BSD-3-Clause license, permitting free use, modification, and redistribution.

The multimodal language models employed are publicly available and used under their respective licenses or terms of service:

- **GPT-5.1** and **GPT-4o**: Proprietary model accessed via OpenAI API under OpenAI's Terms of Service.
- **Claude Sonnet 4**: Proprietary model accessed via Anthropic's API under Anthropic's terms of service.

- **QWEN3-VL-32B INSTRUCT**: Released under the Apache-2.0 license.
- **GLM-4.5V**: Released under the MIT License on Hugging Face.

E Agent Prompts

E.1 Planner agent

The prompt used by the planner agent is shown in Figure 10.

E.2 Circuit Specification Agent

The prompt used by the circuit specification agent is shown in Figure 11.

E.3 Analysis Specification Agent

The prompt used by the analysis specification agent is shown in Figure 12.

E.4 Output Specification Agent

The prompt used by the output specification agent is shown in Figure 13.

E.5 Answer generation agent

The prompt used by the answer generation agent is shown in Figure 14.

You are an expert in planning NGSpice simulations from a circuit schema and a natural language question.

Given the users question, the circuit schema, and relevant domain knowledge, your job is to decide how many NGSpice simulations are required and how the question should be instantiated for each run.

Your task is as follows.

1. If the question asks about a range, change, maximum, or minimum of input values, generate multiple simulations with concrete input values.

2. If the question asks about a range, change, maximum, or minimum of output values, decide whether multiple simulations are needed to capture that variation, and generate them if required.

3. If the question describes a pre switch and post switch scenario, generate two simulations: one for the circuit before the switch and one for the circuit after the switch. Rephrase the question for each run to reflect the corresponding circuit state.

4. If the requested quantity can be obtained using a single NGSpice sweep, such as a DC sweep, parameter sweep, or AC frequency sweep, generate only one simulation and keep the original question unchanged.

5. When multiple simulations are required, clearly specify the number of runs and provide a rephrased question for each run. Preserve all fixed values from the original question and vary only the quantities that are implied to change.

6. Limit the total number of runs to at most 5.

The output format must be followed exactly.

num_runs X
run 1: rephrased question for run 1
run 2: rephrased question for run 2
...
run X: rephrased question for run X

Input will be provided as follows.

Question: {question}
Schema: {schema}
Domain knowledge: {dk}

Output only the number of runs and the rephrased questions in the specified format.

Figure 10: Prompt used for the planner agent

You are an expert in electrical engineering and NGSpice. Your job is to:
1. Update component values in the netlist based on the question requirements
2. Add or correct any missing .model statements for devices that require them

PART 1: Updating Values

Allowed Value Edits

1.1 Update numeric values of existing elements or sources
You may change a numeric literal or parameter expression **only if**:
- The question explicitly gives a value (e.g., "R1 = 2 k", "V1 = 10 V", "C3 = 4 F"), AND
- The name in the question **exactly matches** an element name in the netlist (e.g., R1, V1, Rload).

1.2 Update models of elements when specific parameters are given
For example, if the question specifies a Zener diode with a specific Zener voltage:
.model DZ D(Is=1e-15 N=1 BV=5)

1.3 Introduce frequency parameters when needed
If the question gives a frequency:
- For **Hz**: .param f = <value>
- For **rad/s**: .param f = <value>/(2*3.14159265)

When modifying an AC analysis line, use the parameter:
.ac lin 1 {f} {f}

1.4 Element names must remain type-correct
In NGSpice, device names must begin with a letter corresponding to their device type:
- V voltage source
- I current source
- R resistor
- C capacitor
- L inductor
- D diode
- Q / M transistors

If the question specifies a type that conflicts with the current name, you must **rename** the element.

1.5 Model open circuit elements
If the question specifies an open circuit (e.g., an open resistor), you will set its value to an extremely large number (e.g., 1 e12).
1.6 Type mismatch in schema
If the schema indicates one type but uses another (e.g, a voltage source with the type NPN), correct the type to match the actual device.

Forbidden Value Edits (do not do any of these):
- Do not add or remove any component or source unless explicitly required.
- Do not change values that are not explicitly given in the question.

PART 2: Adding Missing Models
Only the following require models:
- Diodes: D
- BJTs: Q
- MOSFETs: M
- JFETs: J

2.1 General `.model` syntax

    .model <model_name> <device_type> (<parameters>)

Examples:
...

edit:

"""

Figure 11: Prompt used for circuit specification agent

You are an expert in electrical circuit analysis. Given a circuit schema, a question, and domain knowledge, your job is to:

1. Select the correct NGSpice analysis type (DC or AC).
2. Produce a single NGSpice edit specification that configures the netlist accordingly.

Analysis Type Guidelines

Choose **DC analysis** for steady state behavior, including operating point and bias, comparator output, rail saturation or clipping, DC gain or offset, device conduction states, and responses to constant inputs.

Choose **AC analysis** for frequency dependent small signal behavior, including frequency response, gain or phase at a specified frequency, Bode quantities, cutoff or bandwidth, resonance, impedance at a frequency, and any query stated at (f) or (\omega).

DC Analysis Instructions

If DC analysis is needed:

* Convert AC sources to DC sources.
* Set analysis to `.op`.
* Delete any `.ac` or `.tran` statements.
* Remove `.print` and `.save` statements that refer to AC or transient analysis.

AC Analysis Instructions

If AC analysis is needed:

1. Ensure at least one AC stimulus exists by converting a relevant source to:
   `Vx <pos> <neg> AC <value>` or `Ix <pos> <neg> AC <value>`
2. Set analysis to a valid `.ac` statement:
   * If the question specifies (f): use `.ac lin 1 {f} {f}`
   * If the question gives (\omega): set `.param f = /(2*3.14159265)`
3. Delete any `.op` or `.tran` statements unless explicitly required.
4. Replace DC or transient outputs with AC outputs, for example:
   `.print ac V(node) I(source)`

Output Format

Return three parts in order:

1. **Analysis Type**: DC or AC, with a brief justification.
2. **Reasoning**: Step by step edits implied by the chosen analysis.
3. **Edit Specification**: The NGSpice edit specification beginning with `edit:`.

Figure 12: Prompt used for analysis specification agent

NGSpice `.measure` Usage Guide

The `.measure` (or `.meas`) statement instructs NGSpice to compute a **scalar value** from simulation results, such as time, voltage, current, power, peaks, averages, integrals, or event times.

General form

.measure <analysis> <name> <type> <conditions...>

where `<analysis>` {`tran`, `ac`, `dc`} and `<type>` {`WHEN`, `MAX`, `MIN`, `AVG`, `INTEG`, `PARAM`}.

Measurement Types

WHEN (event time)

.measure tran t1 WHEN V(node)=value
.measure tran t2 WHEN I(L1)=0 CROSS=2

Returns the time of the specified event. Optional qualifiers: `CROSS=n`, `RISE=n`, `FALL=n`.

.measure tran vpp PARAM='vmax − vmin'

AC Power Factor and Phase
For power factor, phase angle, or leading/lagging behavior, measure source voltage and current:
.measure ac Vrms RMS V(pos,neg)
.measure ac Irms RMS I(Vsrc)
.measure ac Pavg AVG (V(pos,neg)*I(Vsrc))
.measure ac pf PARAM='Pavg/(Vrms*Irms)'
Use a single frequency point (e.g., `.ac lin 1 {f} {f}`).

DC Operating Point Quantities
For DC quantities such as (V_{CE}) or Q point:
.measure op vce PARAM='V(C)−V(E)'
.measure op ic PARAM='−I(VCC)'
For voltage across a load, always measure `V(load_pos) − V(load_neg)`.

Task Instruction
Given a question and an NGSpice program, decide whether a `.measure` statement is required to answer the question.

* If required, output the appropriate `.measure` statement.
* If not required, explicitly state that no measurement is needed.

Figure 13: Prompt used for output specification agent

You are an expert in electrical circuit analysis and in interpreting NGSpice simulation outputs.

You will receive a circuit schema and diagram, one or more questions, an NGSpice program and its text output, a final question to answer, a reference reasoning trace, and additional domain knowledge.

Your task is to answer the final question using the NGSpice output as the primary source of truth.

Process:

* Interpret the NGSpice output and identify quantities relevant to the question.
* Use NGSpice results even if warnings are present.
* Derive the requested quantity step by step from reported voltages, currents, or phasors.
* Do not compute numeric values manually. All numeric calculations must be performed using the provided tools.
* For equivalent resistance or impedance, always compute it from measured voltage and current in the NGSpice output.
* Preserve polarity and node ordering exactly as stated in the question.
* For AC quantities, use complex values and compute magnitude or phase when required.
* If the required measurement is missing from the output, state that it cannot be determined.
* If comparing two cases, both cases must be present in the output.
* For multiple choice questions, verify each option numerically using the tools and select the correct one.

Output format:

* Show the sequence of tool calls.
* Give a brief explanation if needed.
* Clearly state the final answer.
* If an option letter is requested, output only the letter.

Figure 14: Prompt used for answer generation agent