A Diagnostic Benchmark and Cognitive Scaffolding for Pure Programmatic Reasoning in Large Language Models

Yabo Wang
yabo.wang.cs@gmail.com
University of Auckland

Abstract

Precisely evaluating the intrinsic logical reasoning capability of Large Language Models (LLMs), free from the confounding effects of natural language ambiguity or factual knowledge, is a core challenge in the current field. To address this, we propose an evaluation paradigm called "Pure Programmatic Reasoning" and introduce ARTS (Automated Reasoning Testbed Synthesizer) a novel framework capable of procedurally generating high-fidelity, controllable logic reasoning tests. Our work makes three central contributions. First, we introduce the "Pure Programmatic Reasoning" paradigm and the ARTS framework itself, a novel testbed for generating high-fidelity, controllable logic tests. Second, we propose and validate a "Dynamic Evaluation" methodology that adapts in response to the model's capabilities, demonstrating how to iteratively harden and improve the evaluation framework itself by diagnosing the model's "complex failure modes." Third, through a systematic evaluation of previous state-of-the-art (SOTA) (GPT-4o) and current frontier models (Gemini 2.5 Pro, GPT-5) using this dynamic framework, we reveal a contrasting portrait of the reasoning capabilities of frontier models. We define the frontier models' "Structural Resilience": their remarkable robustness when handling deterministic, algebraically solvable, complex programmatic tasks. Concurrently, we expose their "Structural Fragility": a significant capacity deficit when tasks require processing abstract, non-deterministic logical concepts (e.g., solution spaces), even in the most advanced models. We further quantify a key generational advance through a "Generalization Ablation Study": current frontier models have effectively closed the "Modality Gap" present when processing code, pseudocode, and natural language, demonstrating an abstract logical core independent of syntax, whereas the previous generation exhibited a severe dependence on specific syntax. Finally, we define the boundary of "Cognitive Scaffolding": a synergistic guidance mechanism can effectively fix "execution bottlenecks" caused by high cognitive load. However, we find this standard scaffolding **cannot remedy conceptual deficits** (such as expressing abstract solution spaces), which require specialized diagnostic prompts to resolve. ARTS is designed as a reproducible framework, with plans to make the implementation available to support future research in this area.

1 Introduction

In recent years, Large Language Models (LLMs) have demonstrated extraordinary capabilities across numerous domains [3, 1]. However, the extent to which their success relies on superficial pattern matching learned from vast training corpora versus genuinely intrinsic, generalizable logical reasoning remains an open question [4, 5]. Most existing reasoning benchmarks are coupled with the ambiguity of natural language or the depth of domain-specific knowledge, making the attribution of model failures ambiguous.

To dissect the intrinsic cognitive mechanisms of LLMs, we propose and build a novel evaluation framework –ARTS—designed to create a "Logical Vacuum Chamber **(i.e., a testbed isolated from natural language ambiguity and factual knowledge)**" for measuring the LLM's pure programmatic logical reasoning ability. ARTS is not merely a "Benchmark," but rather a "diagnostic framework" whose goal is to reveal the internal mechanisms of model reasoning. **It is important to note that our goal is diagnostic**—to dissect the internal mechanisms of LLMs—rather than comparatively benchmarking against human performance. Our framework evaluates models against a deterministic, programmatic ground truth (Section 3), not against human performance. This approach is crucial, as human baselines would introduce confounding variables (e.g., calculation errors, working memory limits) that are orthogonal to the pure logical capabilities we aim to isolate. To achieve this diagnostic depth, the framework first creates this "logic vacuum" chamber to isolate pure logical reasoning from confounding factors. This approach enables us to move beyond simple "correct/incorrect" metrics and instead perform the deep mechanistic analysis of model failure attribution that we demonstrate in subsequent sections. Our core contributions include:

(1) Introducing the "Pure Programmatic Reasoning" Paradigm and the ARTS Framework. We introduce ARTS (Automated Reasoning Testbed Synthesizer), a framework capable of procedurally generating controllable logic tests covering 13 scenarios and 4 query modes, thereby eliminating natural language interference and directly assessing the model's core reasoning engine.

(2) Proposing and Validating a "Dynamic Evaluation" Methodology that Adapts in Response to Model Capabilities. We demonstrate how to iteratively harden the evaluation framework itself by diagnosing the model's "complex failure modes." This not only ensures the evaluation's continued effectiveness against frontier model capabilities but also constitutes a methodological contribution on "How we should scientifically evaluate increasingly powerful AI."

(3) Revealing a Contrasting Portrait of Frontier Model Capability Evolution. Through systematic experiments using this framework, we derive five core findings regarding the SOTA models' capability boundaries, including defining their "Structural Resilience" on deterministic tasks, exposing their "Structural Fragility" on abstract logic, quantifying their leap in generalization ability, and defining the boundary of cognitive scaffolding's applicability.

To provide readers with an early overview, our methodology (detailed in Section 3) centers on the ARTS framework, which systematically generates "Anti-Shortcut" test cases based on 13 core reasoning scenarios. It utilizes four distinct "Query Modes" (e.g., deductive, abductive) to probe different cognitive algorithm types. This framework is designed to produce interpretable results through mechanisms like "State Vector Output," allowing for deep diagnostics beyond simple accuracy.

2 Related Work

Our work intersects with three main research areas: LLM reasoning capability evaluation, counterfactual and causal reasoning benchmarks, and code as a reasoning medium.

2.1 LLM Reasoning Capability Evaluation

Evaluating LLM reasoning is an active and extensive research field [16]. Existing benchmarks cover a wide range of tasks, from commonsense reasoning [13], mathematical reasoning [7], to symbolic reasoning [17]. However, many natural language-based benchmarks have been found to suffer from the "shortcut learning" problem [6, 12]. The ARTS framework, through its "Logical Vacuum Chamber" and "Anti-Shortcut" design principles, fundamentally prevents the possibility of exploiting external knowledge and linguistic statistical biases.

2.2 Counterfactual and Causal Reasoning Benchmarks

Counterfactual reasoning is considered central to human causal judgment [9] and has become a essential tool for evaluating LLM robustness [10, 18]. Recently, benchmarks like CounterBench [2] have begun to focus on assessing LLM counterfactual inference ability under formal rules described in natural language. In contrast to these works, ARTS focuses on procedural reasoning: our nested counterfactual mode does not change a single logical premise but procedurally overrides the functions ("laws of physics") and data ("initial conditions") within the code, testing the model's ability to simulate dynamic, ordered state updates.

2.3 Reasoning via Code

Using program code as a medium to evaluate and stimulate LLM intelligence is becoming a new trend [16]. Recent studies show that using Code Prompting can effectively elicit various reasoning capabilities from LLMs [15, 8, 11]. These works typically translate natural language questions into code to aid in solving the original task. The ARTS framework adopts a fundamentally different paradigm. We do not evaluate the model's ability to "write code" or use code as a "chain-of-thought," but rather its ability to "read code" and perform mental simulation. In our "Problem-is-Code" paradigm, the code itself is the task, not a tool for solving another task.

2.4 Distinctions from Prior Research

Our work is distinct from prior research in three key ways: **(1) Target of Evaluation:** We do not evaluate reasoning through code (like Program-of-Thought) but reasoning about code. The code is the problem, creating a "Logical Vacuum Chamber." **(2) Methodology:** We introduce a "Dynamic Evaluation" method where the benchmark itself adapts by diagnosing and adapting to the model's "complex failure modes." **(3) Diagnostic Depth:** The ARTS framework is built for "Result Interpretability," utilizing mechanisms like state vector output (trace) and query mode mirrors (abductive, constrained) to move beyond accuracy and attribute failures to specific cognitive bottlenecks.

3 The ARTS Framework: Designed for Precision Diagnosis

To rigorously validate our findings and ensure that every evaluation allows for unambiguous attribution, we designed and implemented the ARTS framework. All its design decisions serve two methodological cornerstones: Test Purity and Result Interpretability.

3.1 Methodological Cornerstones: Test Purity and Result Interpretability

Our framework design aims to achieve a paradigm shift from "Can the model get the right answer?" to "Can we explain why it got it right or wrong?"

The Single Variable Principle. Every independent test case should be designed to measure one, and only one, core, predefined cognitive capability atom. We strictly avoid coupling multiple independent cognitive challenges within a single test to ensure clarity of attribution.

The Unambiguous Attribution Principle. Test design must ensure that when the model fails, we can attribute the failure to a specific, singular cause with the highest confidence. For example, we proactively exclude the combination of `rule_conflict` (rule conflict) and `constrained` (constraint solving), as this combination would erroneously couple structural judgment and algebraic solving—two different cognitive tasks—leading to ambiguous failure attribution.

3.2 "Anti-Shortcut" Exam Generation Principles

To construct the "Logical Vacuum Chamber" and prevent the model from exploiting statistical shortcuts, every generated

code snippet adheres to a set of strict "Anti-Shortcut" firewall principles:

Non-Informative Logic Naming. All variables and function names participating in the core logic must be randomly generated, nonsensical names (e.g., ZIKLO, BLAF) to prevent the model from guessing their function through their name.

Minimum Computational Load. Numeric operations are restricted to simple addition, multiplication, and subtraction with minimal computational complexity. This principle separates the mental execution capacity of logical flow (our measurement target) from pure arithmetic computational ability (not our focus), isolating the independent variable.

Random Number Anti-Memorization. All numerical constants appearing in the code are randomly generated and do not follow typical conventions (e.g., step sizes of 2, 5, 10) to prevent the model from guessing via memorized patterns.

Transparent and Deterministic Logic. The inherent logical structure of the code must be fully deterministic, meaning given valid inputs, the output is uniquely determined. This prevents model failure attribution from being muddied by the task itself containing logical ambiguity.

Controlled Value Range. To establish a pure logic capability baseline during the initial benchmarking phase, we follow the "Minimum Computational Load" principle, setting the baseline numerical range to single digits (e.g., 2-9). It is noteworthy that `value_range` is a fully configurable parameter in the framework, and in subsequent stress testing phases, we systematically adjust it to larger ranges (e.g., 100-999) to evaluate the model's computational fidelity.

Sufficient Logical Depth. Through the configurable `num_vars` parameter, we ensure the calculation chain has sufficient logical depth. In baseline testing, we typically set this to 4 or 5 steps to break simple input-output pattern matching and force the model to perform step-by-step reasoning. In stress testing, this parameter is systematically increased to probe the model's capacity boundary for handling long-range dependencies.

Data-Layer Distraction. To test attentional filtering, the `num_distractors` parameter injects irrelevant "data noise" (i.e., unused variables and calculations) into the code's scope, distinct from the more complex structural-level logical noise (e.g., unused functions) `meta_distractor_level` introduced in later stress tests.

3.3 Task Generation Mechanism: "One World, Multiple Questions"

The mechanism for task generation is central to our "Dynamic Evaluation" methodology and the scientific rigor of our framework. The core of our approach is to programmatically implement the "One World, Multiple Questions" principle, ensuring that all comparisons are fair and controlled. This mechanism works in harmony with the "Single Variable Principle" (Section 3.1): the "Logic World" (generated in Stage 1) serves as the common, controlled logical foundation, while the "Multiple Questions" (generated in Stage 3) become the individual, independent *test cases*. Each of these "mirror exams" is designed to probe a distinct cognitive capability (e.g., forward simulation fidelity vs. inverse algebraic solving), thereby isolating the specific cognitive atom under evaluation.

This pipeline, orchestrated by `test_case_generator.py`, proceeds in four distinct stages (illustrated in Figure 1):

Stage 1: Logic World Generation (`scenarios.py`). For any given seed, the corresponding template in `scenarios.py` (e.g., CausalChainTemplate) is invoked. Its job is to generate a single, syntax-agnostic "logical blueprint" called an **abstract_logic** – a representation that captures the pure logical structure independent of any programming language. This dictionary defines the entire "physical law" of a unique logical world: its structure (e.g., a 5-step chain), its operations (e.g., +, *), and its specific numerical constants (e.g., + 5, * 2).

Stage 2: Parallel Query Generation (`generators.py`). This single, unified `abstract_logic` is then passed in parallel to the four different query generators within `generators.py`. Each generator's job is to "reshape" this same logical world into a different cognitive task.

Stage 3: Creating "Mirror Exams". The Deductive generator creates a standard forward-simulation task (a "Deductive Reasoning" script). The Abductive generator takes the exact same logic but hides the initial input and uses the final answer (calculated in Stage 4) as a known condition, creating an "Abductive Reasoning" script. The Constrained generator uses this logic as "Rule A," programmatically generates a "Rule B" from it, and asks the model to find the `param_X` that makes their outputs match. The Nested CF generator likewise uses this logic to define "Rule A" and "Rule B," creating a dynamic state-update simulation.

Stage 4: Ground Truth Generation (`solvers.py`). In parallel with Stage 2, the `abstract_logic` is also sent to `solvers.py`. The solver, acting as a perfect oracle, executes the logic for each required mode (e.g., forward for deductive, backward for abductive) to generate the deterministic `golden_answer`.

This pipeline guarantees that when we compare a model's performance on a deductive task versus an abductive task, we are testing its ability to reason (forward vs. backward) about the exact same underlying logical problem, thus eliminating any confounding variables.

3.4 13 Core Designed Scenarios

Our framework includes 13 meticulously crafted reasoning scenarios, each targeting a specific cognitive primitive, as detailed in Table 1.

3.5 Query Modes and Difficulty Console

To achieve in-depth diagnosis of the model's cognitive abilities, we built a systematic "Difficulty Console." This console fundamentally changes the *type* of cognitive algorithm required by probing two distinct categories of reasoning, which we must first define:

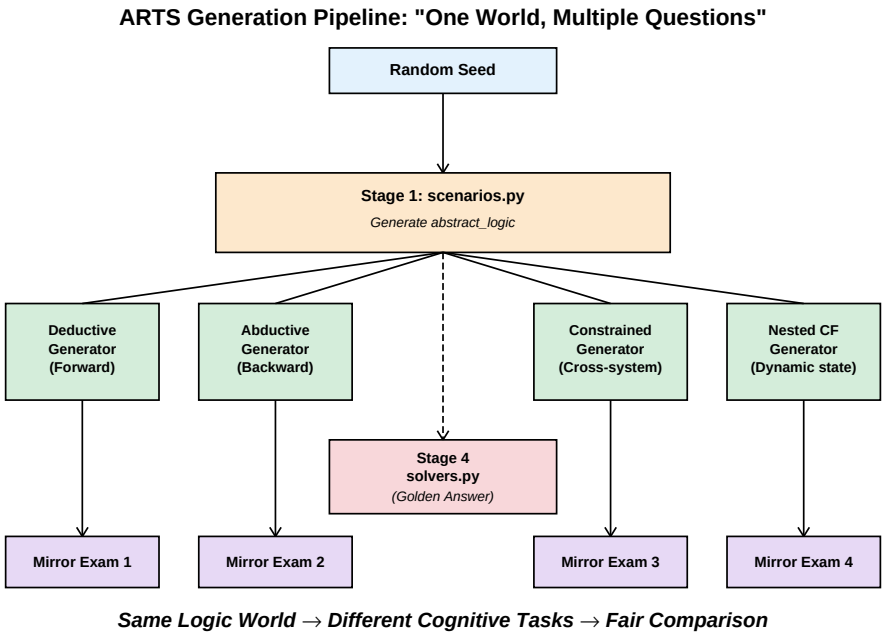

Figure 1: ARTS Generation Pipeline illustrating how a single random seed flows through `scenarios.py` to generate an `abstract_logic` (logical world), which is then used by four different generators in parallel to produce "mirror exams" for fair comparison.

- **Deterministic Reasoning:** Executing a known computational path to find a single, unique numerical answer (e.g., $x = 5$).
- **Abstract Logical Reasoning:** Processing non-deterministic, conceptual-level logic, such as a **"solution space"**—the set of all possible inputs that satisfy a given condition (e.g., any value where $x \neq 0$).

Our framework applies pressure across these reasoning types using four core query modes and multiple orthogonal difficulty dimensions.

Cognitive Ladder: Four Core Query Types We follow the scientific principle of "one world, multiple questions." For any "logical world" determined by a specific random seed and configuration, we generate a set of logically strongly correlated "mirror exams," each corresponding to one of the four increasingly difficult query modes below:

deductive (Deductive Reasoning). This is the most basic forward simulation task: "Given all causes and rules, solve for the unique result." It aims to assess the model's most fundamental programmatic logic execution capability and reasoning fidelity, serving as the performance baseline for all tests.

abductive (Abductive Reasoning). This is the inverse of deduction: "Given rules, partial causes, and the final result, solve backward for an unknown initial condition." It aims to assess the model's programmatic inverse reasoning capability, which carries a much higher cognitive load than deduction.

constrained (Constraint Reasoning). This is a more advanced, cross-system abductive reasoning, requiring the model to solve for an unknown parameter such that the output of one system with a specific input is exactly the same as the output of another reference system.

nested_counterfactual (Nested Counterfactuals). This is the ultimate test of the model's mental flexibility and state update capacity. The task requires the model to first handle a change in the world's "laws of physics" (rule override), then handle a change in the "initial conditions" (data override), and finally perform the calculation under the new rules and new data.

To demonstrate how these four query modes are programmatically constructed from the same underlying logic, we provide examples of the core code generation templates in Appendix D.

Orthogonal Difficulty Control Dimensions We apply controllable pressure to the model across multiple dimensions through a series of configurable parameters. In the benchmarking phase, these parameters are set to "low-pressure" baseline values (e.g., `num_vars=4`, `num_distractors=0`) to establish a performance baseline; in the stress testing phase, they are systematically raised to probe the capability boundaries.

Ultimate Stress Test: Cognitive Challenges to Explore Capability Boundaries In addition to the systematically adjustable difficulty dimensions mentioned above, the ARTS framework includes an "Ultimate Stress Test" suite custom-designed for top-tier models, aimed at exploring their capability ceiling even when they perform perfectly under conventional high pressure. This suite includes:

Meta-Logical Distraction: Injects complex, syntactically cor- rect, but logically irrelevant "dead code" (e.g., uncalled func- tions) to test the model's ability to identify and prune irrelevant reasoning paths.

Algorithmic Pattern Recognition: Embeds simple algorith-

Table 1: 13 Core Reasoning Scenarios in the ARTS Framework

Category	Scenario Name	Core Capability Tested
1. Fundamental Causal Structures	1. Direct Cause-and-Effect	Most basic single-step causal inference.
	2. Causal Chain	Ability to maintain reasoning fidelity in a long, linear logic chain.
	3. Common Effect / Collider	Ability to process converging logic of "multiple causes to a single effect."
	4. Common Cause / Fork	Understanding of branching logic where a single cause leads to multiple independent effects.
2. Compound Causal Logic	5. Conditional Causality	Understanding and executing branching logic of if-else structures.
	6. Causal Inhibition	Understanding how an inhibitory condition "shuts down" or alters the main causal path.
	7. Redundant Causality	Reasoning when multiple causes are independently sufficient to lead to the same result.
3. Dynamic & Modulatory Logic	8. State-Dependency	Reasoning when the calculation result depends on the system's prior state.
	9. Feedback Loop	Ability to handle self-updating values and cumulative effects in iterative calculation.
4. Structural & Hierarchical Logic	10. Hierarchical Reasoning	Understanding and executing nested or layered function/-module call logic.
	11. Ignoring Irrelevant Info	Ability to identify and focus on the core logic chain when there are numerous irrelevant variables and calculation paths.
5. Constraint & Boundary Logic	12. Boundary Conditions	Reasoning when the calculation result depends on whether a certain threshold is exceeded.
	13. Rule Conflict	Ability to judge and execute the correct logic branch when a logical conflict (e.g., try-except) exists.

mic patterns (e.g., finding a least common multiple) within the code to test if the model performs mechanical simulation or recognizes the higher-level conceptual "shortcut."

Structural Composition Pressure: Creates nested hierarchi- cal tasks where the main logic must call and correctly interpret the results of sub-modules, testing the model's "mental call stack" and state management capabilities.

3.6 Evaluation Paradigm and Diagnostic Mechanisms

Closed-Book Exam. The code submitted to the LLM does not contain any answer clues. All validation is performed in the background by comparison with the `golden_answer` generated by `solvers.py`.

Instruction Following as a Core Evaluation Dimension. The strict requirement for the model's output format is not a technical detail but a core design. It measures a higher-level capability: whether the model can precisely follow instructions and constraints while executing complex reasoning.

3.6.1 Procedural Scaffolding: The Dual Role of the Trace Mechanism

At the core of our framework, the `trace` mechanism is not merely a technical detail for data collection; it is itself a key methodological tool, playing the dual role of a diagnostic probe and a cognitive scaffold.

State Vector Output for Diagnosis (as a Diagnostic Probe). In implementation, `trace` is a Python dictionary. Our code generator (`scenarios.py`) procedurally inserts an assignment statement, such as `trace['VAR_B'] = val_var_b`, after every critical calculation step. This requires the model, upon task completion, to output a "state vector" containing the values of all key intermediate variables throughout the calculation process.

Code-Embedded CoT as Cognitive Scaffolding (as a Cognitive Scaffold). More importantly, the act of requiring the model to fill in the `trace` dictionary is, in itself, a powerful cognitive intervention. We call this design "Code-Embedded Chain-of-Thought," and it plays the role of "Procedural Scaffolding."

3.6.2 Internal Consistency Guarantee: The Self-Validation Closed Loop of Test Cases

To ensure the high reliability of the benchmark, the ARTS framework integrates a critical Self-Validation Closed Loop mechanism. This mechanism ensures that every ultimately generated test case is logically sound, solvable, and unambiguous.

4 Experimental Validation and In-Depth Analysis

To systematically dissect the model's reasoning capabilities, we designed a dynamic, multi-stage experimental blueprint.

4.1 Advanced Metrics: Beyond Accuracy

To address the need for "more in-depth analysis" and move beyond a single accuracy score (Acc), we introduce two advanced, context-aware metrics derived from the ARTS framework's unique diagnostic capabilities.

4.1.1 Metric 1: Failure Attribution (Qualitative)

Simple accuracy treats all failures as equal. However, our diagnostic analysis, enabled by the "Dynamic Evaluation" methodology, reveals that not all failures are the same. We formally categorize failures into two distinct types, and this classification itself is a key finding for understanding model limitations:

Execution Bottlenecks: These are failures where the model demonstrates a correct logical understanding of the task (e.g., in its reasoning trace) but fails during the procedural computation. This is typically due to high **cognitive load**, which we **qualitatively** define as the mental effort required to maintain and update computational state over a long series of steps (e.g., losing track in a long calculation chain). We term this failure a "bottleneck" in its reasoning fidelity. As we will show in Section 5.4, these failures are often fixable with "Cognitive Scaffolding.'

Core Capability Deficits: These are fundamental failures where the model does not possess the underlying conceptual or logical tools to even begin solving the problem, regardless of scaffolding. As shown in Section 5.3, the inability to handle abstract, non-deterministic "solution spaces" is a prime example of such a deficit, which cannot be fixed by simply optimizing the calculation steps.

This qualitative metric allows us to distinguish between a model that knows what to do but fails in doing it, versus a model that doesn't know what to do at all. This is essential for pinpointing the true capability boundary.

4.1.2 Metric 2: Process Accuracy (Quantitative)

Traditional accuracy (Accuracy) only focuses on whether the final answer is correct or wrong, which is an "endpoint metric." This method cannot distinguish between "complete failure with an error in the first step" and "near success with an error only in the last step."

To more finely quantify the model's reasoning ability, we introduce "Process Accuracy" as a quantitative metric. This metric utilizes the core mechanism of the ARTS framework—"State Vector Output" (i.e., the trace dictionary).

Definition: For a test case that ultimately fails, its "Process Accuracy" is defined as the percentage of intermediate steps that the model correctly calculated in the trace dictionary.

For example, in a `causal_chain` task with 10 intermediate steps, if the model correctly completes the first 8 steps but makes an error in step 9 leading to an incorrect final answer, its "endpoint accuracy" is 0%, but its "process accuracy" is 80%.

This metric enables us to quantify the model's "Reasoning Fidelity"—the extent to which the model can maintain accuracy of its computational state in complex long-chain reasoning. It provides us with an analysis tool that is far richer and more diagnostic than binary (correct/incorrect) accuracy. It is important to note that this specific metric, based on the code-embedded `trace` dictionary, is primarily designed to measure the simulation fidelity of `deductive` (forward) reasoning tasks. For inverse reasoning modes like `abductive` and `constrained`, the model's reasoning steps are captured in the output `reasoning` key, not the `trace` dictionary. A full 'Process Accuracy' evaluation for these modes would require automated parsing and logical validation of that reasoning key, which we delineate as a key area for future work.

4.2 Stage One: Baseline Diagnosis

Research Question: How does the model's "Structural Reasoning Fidelity" perform in a low-pressure environment? Where are the capability bottlenecks?

Experimental Design: We designed a systematic 2x2 experimental matrix for the Stage One baseline diagnosis, aimed at comprehensively covering two core types of structural pressure.

4.2.1 Baseline Diagnosis Matrix

To systematically assess the model's foundational performance under different types of structural pressure, we test each reasoning scenario within a 2x2 "World Complexity" matrix. This matrix is composed of two orthogonal complexity dimensions:

Intervention Complexity. This dimension, controlled by the `input_vars` parameter, aims to test the model's ability to handle different numbers of concurrent inputs.

Atomic Intervention: `input_vars = 0`. In this configuration, each scenario only handles its most fundamental core inputs, corresponding to a test of a "single causal path."
Multi-point Concurrent Intervention: `input_vars > 0`. In this configuration, additional, parallel input variables are introduced into the scenario, testing the model's ability to handle "multi-cause-single-effect" or "multi-path interference."

System Dynamics. This dimension, controlled by the `enable_rule_intervention` parameter, aims to test the model's reasoning ability under static versus dynamic rules.

• *Single Rule / Static Physics:* `enable_rule_intervention = False`. The model only needs to reason under one fixed set of "laws of physics."

• *Rule Intervention / Dynamic Physics:* `enable_rule_intervention = True`. Two sets of functionally different but structurally similar rules (Rule A and Rule B) are defined in the code, and a `rule_switch` variable dynamically decides which set of rules to use at runtime. This requires the model to have the ability to switch its mental model based on context.

By combining these two dimensions, we generate four basic complexity configurations for each reasoning scenario, forming a comprehensive baseline diagnostic matrix.

Based on the results of the Stage One baseline diagnosis, we partition all (model-scenario-query mode) combinations into three zones: **Group A** (Main Battleground, 15%-85% accuracy), **Group B** (High-Performance Zone, >85%), and **Group C** (Performance Collapse Zone, <15%), to guide subsequent targeted testing.

4.3 Stage Two: Targeted Stress Testing

Research Question: Under high-pressure protocols, to what extent can cognitive guidance enhance or maintain the model's "resilience"?

Experimental Design: For the Groups A, B, and C identified in Stage One, we adopt a differentiated "High-Pressure Prompt Comparison Experiment" protocol. For example, for representative scenarios in Group A, we independently or jointly push their core pressure parameters (e.g., `num_vars`, `value_range`) to the limit and compare the performance difference between the **Standard Task Definition Prompt** (see Appendix C.2) and the **Final Solution / Cognitive Scaffolding Prompt** (see Appendix C.3).

4.4 Stage Three: Ultimate Stress Test and Deep Diagnosis

Research Question: For top-tier models that still perform perfectly under conventional high pressure, where will higher-dimensional cognitive complexity tear open the crack in their capability?

Experimental Design: We activate the "Ultimate Stress Test" suite, introducing higher-dimensional challenges such as Meta-Logical Distraction, Algorithmic Pattern Recognition, and Structural Composition Pressure, as defined in Section 3.5.

4.5 In-Depth Validation: The 2x2 Factorial Experiment of Guidance Effects and Interaction Analysis

To move beyond isolated studies of single guidance methods and systematically dissect the relationship between different cognitive support mechanisms, we designed and executed a 2x2 factorial experiment. The core objective of this experiment is to simultaneously and orthogonally assess the main effects of two core guidance mechanisms—"Procedural Scaffolding" and "Metacognitive Guidance"—and their interaction effect.

Experimental Design: We selected the `causal_chain | constrained` task and used the "Multi-point Concurrent - Rule Intervening" high-pressure configuration defined in Section 4.2. We chose `num_vars=8` as the stress point, as prior testing (see Figure 3) has demonstrated that GPT-4o encounters a "Sharp Performance Decline" around this complexity level.

We defined two core experimental factors, each with two levels:

Factor A: Procedural Scaffolding. High (`trace=True`): Generated code includes `trace[...]` statements, requiring the model to output intermediate states step-by-step. Low (`trace=False`): Generated code does not include `trace[...]` statements, requiring the model to output only the final answer.

Factor B: Metacognitive Guidance. Standard (**Standard Prompt**): Uses the **Standard Task Definition Prompt** (see Appendix C.2), defining only the task goal and format. Strong Guidance (**FUSION Prompt**): Uses the **Final Solution / Cognitive Scaffolding Prompt** (see Appendix C.3), additionally providing detailed metacognitive problem-solving strategies.

By evaluating these four combinations (`High/Standard`, `High/FUSION`, `Low/Standard`, `Low/FUSION`), we are able to conduct in-depth attribution analysis of the root causes of model failures (e.g., lack of algorithm, lack of working memory, or both) in the results section (Section 5).

4.6 Prompt Hierarchical Design: A Multi-Level Cognitive Diagnostic Matrix

Our experimental design relies on a layered Prompt system (detailed in Appendix C), where each level of Prompt plays a different diagnostic role, aimed at dissecting the model's reasoning process progressively from general to specific. This system forms a complete "Cognitive Diagnostic Matrix," allowing us to precisely separate and measure different cognitive capabilities.

Layer One: General Baseline. We use a unified **General Baseline Prompt** (see Appendix C.1), which requires the model to act as a "Pure Logic Processor," abandoning all prior knowledge and working only as a code simulator. This Prompt is used for the Stage One baseline diagnosis, aiming to establish a performance benchmark with minimum metacognitive guidance.

Layer Two: Standard Task Definition. For more complex query modes like `abductive` and `constrained`, we provide task-specific standard guidance Prompts (e.g., the **Standard Task Definition Prompt**, detailed in Appendix C.2). These Prompts clearly define the task goal and output format, constituting the "Standard Control Group" in high-pressure testing.

Layer Three: Metacognitive Guidance (FUSION Prompts). This is our designed Prompt series (e.g., the **Final Solution / Cognitive Scaffolding Prompt**, shown in Appendix C.3) aimed at providing the strongest cognitive guidance, forming the "Experimental Group" in high-pressure

testing. Its core design is the synergistic embodiment of "Cognitive Scaffolding": (1) Forcing process externalization through the `reasoning` key; (2) Injecting metacognitive strategies through examples. It is critical to differentiate this `reasoning` key from the code-embedded `trace` dictionary: the `trace` dictionary (Section 3.6) captures the model's fidelity in simulating the code's state, while the `reasoning` key captures the model's fidelity in executing the abstract problem-solving algorithm required for inverse-reasoning tasks.

Layer Four: Advanced Diagnostic Probes. To explore the boundaries of capability, we also designed specialized probes. For example, the **Abstract Logic / Solution Space Probe Prompt** (see Appendix C.4) explicitly instructs the model on how to report a non-deterministic "solution space," enabling us to diagnose whether the model's "Structural Fragility" stems from a core capability deficit or merely the lack of instructions to express abstract concepts.

4.7 Models and Evaluation

Models. All experiments in this study were conducted on GPT-4o, Gemini 2.5 Pro, and GPT-5, with the temperature parameter set to 0 to ensure result reproducibility.

Evaluation Process. Automated API calls and deep comparison using the `compare_results` function. Each experimental condition was conducted on 100 independent test cases, generated using 100 unique random seeds..

5 Experimental Results and Analysis: A Portrait of Frontier Model Reasoning Capabilities

Through systematic experiments, from low-pressure baseline to extreme stress testing, we reveal five core findings about frontier model reasoning capabilities. Together, these findings paint a complex and profound portrait of capabilities and drive our subsequent experimental design and framework iteration.

We conducted a series of systematic evaluations on the previous top-tier model (GPT-4o) and current frontier models (Gemini 2.5 Pro, GPT-5) using the ARTS framework.

5.1 Stage One: Baseline Diagnosis Reveals a Generational Divide

We began by running the Stage One: Baseline Diagnosis (defined in Section 4.2) on all three models: GPT-4o, Gemini 2.5 Pro, and GPT-5. The results immediately revealed a stark difference in capabilities between the model generations:

The frontier models (Gemini 2.5 Pro and GPT-5) exhibited near-perfect performance, demonstrating strong robustness across all four baseline complexity configurations (Atomic/Multi-point, Static/Dynamic).

In sharp contrast, the previous-gen model (GPT-4o) was highly vulnerable **under standard guidance**. While it handled simple configurations, its accuracy collapsed under the high-pressure 'Multi-point Concurrent - Rule Intervening' configuration, particularly in inverse-reasoning scenarios like `causal_chain | constrained`.

This baseline finding is the cornerstone of our entire analysis. It dictates the two divergent paths we take in the subsequent sections:

(1) For the frontier models, their baseline perfection demonstrated that 'Stage One' tests were insufficient. We were forced to immediately escalate to **Stage Two (Targeted Stress Testing)** and, subsequently, **Stage Three (Ultimate Stress Test)** to find their capability boundaries, which we analyze in detail in Section 5.3.

(2) For the previous-gen model, its clear baseline bottlenecks identified GPT-40 and the `causal_chain` task as the ideal subjects for our deep dive into "Sharp Performance Decline" (Section 5.5) and the mechanics of scaffolding (Section 5.7). This baseline perfection in frontier models led us directly to our first finding: defining the upper bounds of this capability in the face of extreme complexity.

5.2 Finding One: Structural Resilience: Defining the Frontier of Deterministic Reasoning (GPT-5 & Gemini 2.5 Pro)

Our first core finding is the remarkable robustness exhibited by current frontier models when handling deterministic, algebraically solvable, complex Pure programmatic reasoning tasks, which we define as "Structural Resilience."

As the benchmark for our evaluation, GPT-5 demonstrated perfect (100%) in almost all tests, setting a "Gold Standard" for the capability ceiling of current SOTA models in this area.

This resilience held true across both Stage One and Stage Two testing. In the **Stage One** baseline diagnosis, GPT-5 achieved 100% accuracy across all 13 scenarios and 4 query modes. We then proceeded to **Stage Two: Targeted Stress Testing** by pushing the logical depth (`num_vars`) to its limit. As shown in **Figure 2(c)**, GPT-5's accuracy remained a perfect 100% even as the logical depth increased to 25 steps.

Because both Stage One and Stage Two failed to find a capability ceiling, we escalated to the **Stage Three: Ultimate Stress Test** (see Section 4.3). More critically, even when facing our custom-designed `hierarchical_reasoning | constrained` task (see Appendix B.1) from this suite—which includes quadruple cognitive pressure (structural depth, non-linear logic, high computational load, and strong distraction)—GPT-5's accuracy **still** maintained 100%.

5.3 Finding Two: Structural Fragility: Exposing the Common Deficit of Frontier Models in Abstract Logic (GPT-5 & Gemini 2.5 Pro)

While Section 5.2 demonstrated frontier models' remarkable robustness on deterministic tasks, a natural question arises: does this resilience extend to tasks requiring abstract logical reasoning? Our next finding reveals a striking contrast.

"Structural Fragility" is the failure to handle this second, more abstract type of reasoning.

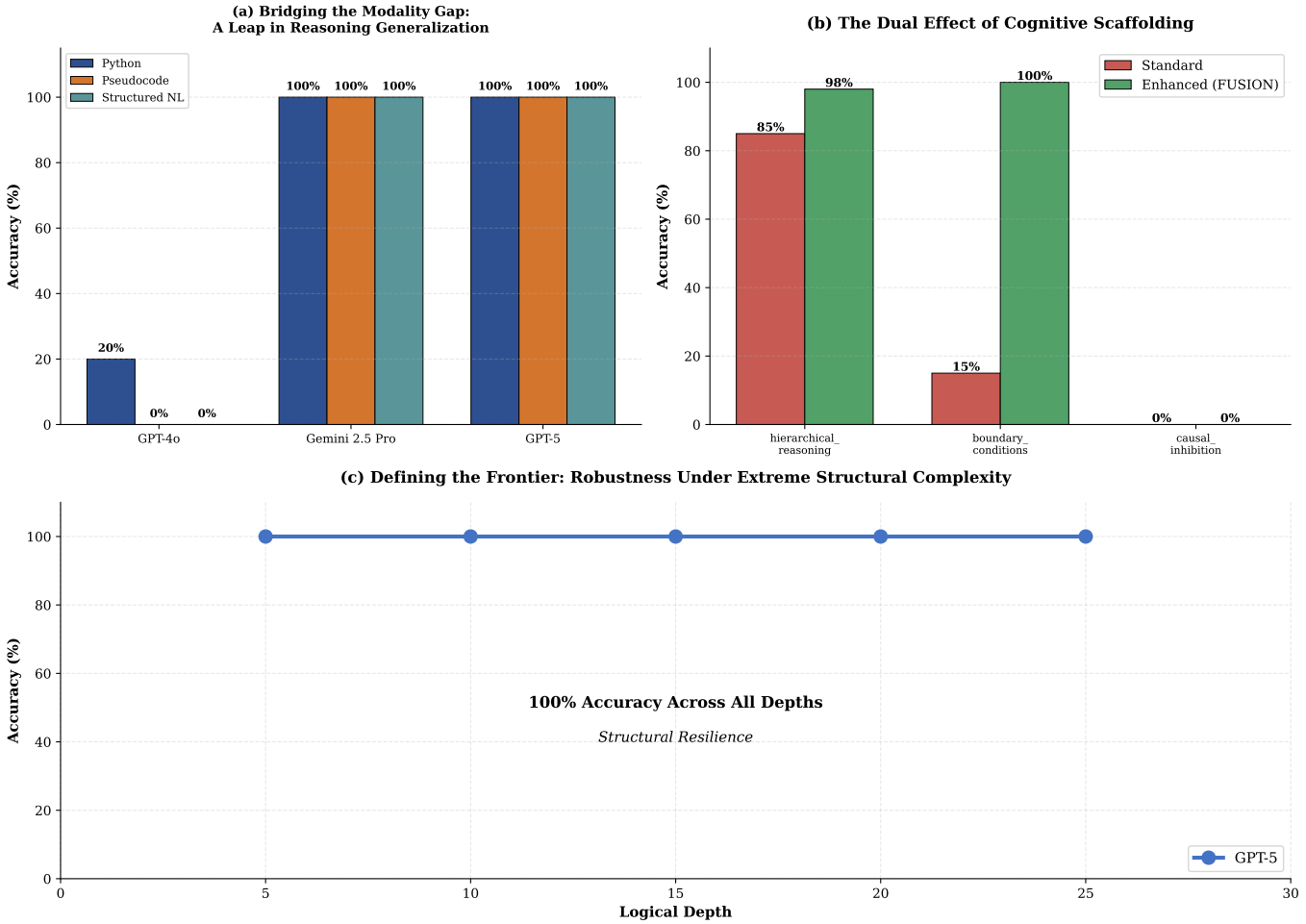

Figure 2: The Binary Portrait and Evolutionary Trajectory of Frontier Model Reasoning Capabilities. These three subfigures collectively illustrate the key findings: (a) shows the dramatic difference between GPT-4o's "cliff" (20%, 0%, 0%) and frontier models' perfect performance (100% across all modalities); (b) demonstrates how cognitive scaffolding fixes execution bottlenecks (boundary_conditions: 15%→100%) but cannot address core deficits (causal_inhibition: 0%→0%); (c) shows GPT-5's perfect structural resilience under extreme complexity (100% across all depths from 5 to 25).

In sharp contrast to their **remarkable** robustness in deterministic tasks (which we define as "Structural Resilience"), when tasks required processing these *abstract logical concepts* (as defined in Section 3.5), even the most frontier models like GPT-5 and Gemini 2.5 Pro collectively exposed a **significant** "Structural Fragility."

This anomalous result prompted us to initiate a **systematic failure analysis** of the failure cases. We found that this systemic failure did not stem from the model's calculation error or logical confusion; on the contrary, it revealed a higher-order metacognitive insight we had not initially anticipated, while simultaneously exposing a fundamental deficit in its capacity to express abstract concepts.

To validate this hypothesis, we conducted a follow-up test using the specialized Abstract Logic / Solution Space Probe Prompt (detailed in Appendix C.4). This prompt explicitly instructs the model on how to formulate and report a non-deterministic 'solution space' (e.g., $x \neq 0$) instead of a single integer. When this prompt was applied to the same `causal_inhibition` tasks, the performance of both Gemini 2.5 Pro and GPT-5 surged from 0% to over 90%. This confirms that the "Structural Fragility" is not a core *reasoning* deficit, but rather a profound *conceptual-expressive* deficit—a **failure of expression** where the model's native output mechanisms cannot handle the abstract concepts (like solution spaces) that its own reasoning engine correctly infers. Our "Dynamic Evaluation" methodology successfully diagnosed and remedied this expressive failure.

5.4 Finding Three: The Boundary of Cognitive Scaffolding: Fixing Execution Bottlenecks, but Not Conceptual Deficits

Our third core finding is the clear demarcation of the "Cognitive Scaffolding" boundary. We demonstrate this by contrasting the effect of our **standard FUSION Prompt** on two different types of failures: `boundary_conditions` and `causal_inhibition`.

Fixing the "Execution Bottleneck" (Gemini 2.5 Pro). For `boundary_conditions`, a purely deterministic long-

range inverse reasoning task, Gemini 2.5 Pro's failure under standard guidance (15% accuracy) was primarily due to an "Execution Bottleneck" under high cognitive load. As shown in Figure 2(b), our standard FUSION Prompt (which provides a step-by-step *calculation* algorithm) perfectly fixed this, restoring performance to 100%.

Exposing the "Conceptual Deficit" (Gemini 2.5 Pro & GPT-5). Crucially, the **same standard FUSION Prompt** was powerless when facing the abstract logic challenge of `causal_inhibition`, with both Gemini 2.5 Pro and GPT-5's accuracy remaining at 0% (Figure 2(b)). This does not contradict our finding in Section 5.3; it **reinforces** it. It proves that this failure is not an "Execution Bottleneck," but a deeper "Conceptual Deficit." The model failed not because it couldn't *calculate*, but because our standard prompt lacked the *conceptual vocabulary* to express a "solution space." Only the specialized **Solution Space Probe** (as discussed in Section 5.3) could provide this missing vocabulary. This contrast perfectly defines the boundary of our standard cognitive scaffolding.

5.5 Quantifying Scaffolding's Protective Effect Against Cognitive Load

Finally, by turning our diagnostic lens onto the previous top-tier model, GPT-4o, we precisely quantified the essential role of cognitive scaffolding in defending against the cognitive load imposed by increasing logic chain length.

To do this, we conducted a high-pressure test on the `constrained` mode of the `causal_chain` sce-nario, specifically using the foundational 'Atomic Inter-vention Static Physics' configuration (as defined in Section 4.2). We systematically increased the length of the logic chain (`num_vars`) and compared the model's performance under "Standard Guidance" and "Strong Guidance."

This analysis focuses specifically on GPT-40, as the frontier models (Gemini 2.5 Pro and GPT-5) achieved near-perfect performance on this task, demonstrating no significant "performance decline" to analyze.

The results (shown in Figure 3) clearly reveal two drastically different performance decay patterns:

"Sharp Performance Decline". Under Standard Guidance, GPT-4o's performance collapsed sharply as the logic chain grew. At 3 steps, its accuracy was only 55%; when the chain increased to 5 steps, performance plummeted to 15%; and at 8 steps and above, the model completely failed, with 0% accuracy.

Graceful Degradation. In sharp contrast, under our FUSION Prompt (Strong Guidance), GPT-4o's reasoning resilience was fundamentally enhanced. The model maintained an extremely high accuracy of 90% even with a logic chain up to 12 steps. Its performance only began to show a noticeable decline at 15 steps, but still remained at a serviceable 65% level.

Through our qualitative analysis of the model's `reasoning` key (detailed in Appendix B.3), we can clearly see how this mechanism operates. Under standard guidance (Figure 3, orange line), when the model's performance degraded at `num_vars=3` (55% accuracy), its `reasoning` key (as shown in Appendix B.3.1) exposed that it attempted to rely on unreliable "intuition," leading to context confusion. However, under strong guidance (Figure 3, blue line), the model maintained extremely high accuracy at `num_vars=15`. As shown in Appendix B.3.2, the model's `reasoning` key proves that it had completely abandoned "intuition" and instead faithfully and precisely executed the 15-step reverse solving algorithm we provided in the Prompt on its "scratch paper" (the reasoning key).

This finding eloquently proves that the core value of cognitive scaffolding is not only to improve basic accuracy but, more fundamentally, to change the model's failure mode—transforming an unpredictable "Sharp Performance Decline" into a foreseeable "Graceful Degradation." It demonstrates that the success of the FUSION Prompt comes from replacing the model's unreliable "internal algorithm" (from mental calculation) with a reliable "external algorithm" (from the Prompt), thereby fixing the "Execution Bottleneck." By forcing the model to "externalize thought" and follow a reliable procedural algorithm, cognitive scaffolding greatly extends the boundary of complexity for reasoning tasks that the model can reliably execute.

Having analyzed the model's resilience against *structural complexity* (i.e., the length of the logic chain) in the previous section, we next investigate its robustness across a different dimension: *modality*. This "Generalization Ablation Study" (Finding Four) was designed to answer whether the model's reasoning capability is tied to Python syntax or if it represents a more abstract, generalizable logic.

5.6 Finding Four: Closing the Modality Gap—The Leap in Reasoning Generalization

To ultimately verify the extent to which the model's reasoning capability is an abstract, generalizable ability independent of specific syntax, we designed and executed a "Generalization Ablation Study." We selected the `hierarchical_reasoning | constrained` task, which carries the highest cognitive load, and logically equivalently rendered it into three forms: Python code, pseudocode, and structured natural language.

GPT-4o's "Generalization Cliff". The performance of the previous top-tier model, GPT-4o, clearly exposed the limitations of its capabilities. First, even on the Python code it is most familiar with, the low accuracy of 20% indicates that the high-complexity task already caused a severe "execution bottleneck."

The Abstract Logical Core of Frontier Models. In sharp contrast, GPT-5 and Gemini 2.5 Pro both achieved **perfect (100%)** accuracy across all three formats. This result is highly significant: it indicates that these two models have successfully decoupled their underlying abstract logical reasoning capability from the specific form in which it is presented.

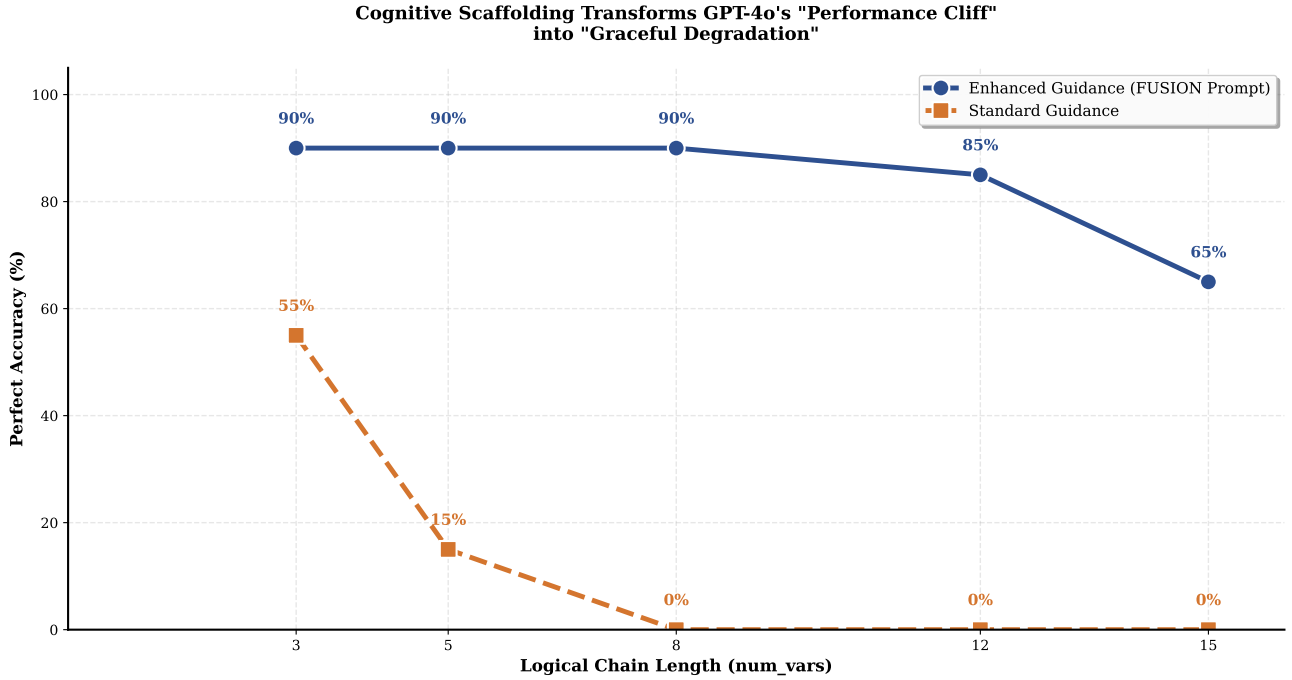

Figure 3: Cognitive Scaffolding Transforms GPT-4o's "Sharp Performance Decline" into "Graceful Degradation". This figure demonstrates the transformative effect of cognitive scaffolding on GPT-4o's reasoning resilience across increasing logical chain lengths in the `causal_chain | constrained` task. **The experiment uses the foundational 'Atomic Intervention - Static Physics' configuration (defined in Section 4.2).** Under standard guidance (orange dashed line), performance exhibits a "Sharp Performance Decline", dropping from 55% at 3 variables to 15% at 5 variables, and completely collapsing to 0% at 8 variables and beyond. This sharp decline proves that without external support, the model's internal "working memory" cannot maintain fidelity for long-range reasoning. Under enhanced guidance with the FUSION Prompt (blue solid line), the model maintains exceptionally high performance (90%+) up to 12 variables and degrades gracefully thereafter, reaching 65% at 15 variables. This transformation from a sharp decline to graceful degradation proves that scaffolding fundamentally changes the failure mode by forcing externalization of computational state, thus greatly extending the boundary of complexity for reasoning tasks the model can reliably execute.

5.7 Core Finding Five: The Synergistic Effect of Cognitive Scaffolding

The analysis in Section 5.5 (Figure 3) demonstrated *that* cognitive scaffolding is effective, transforming a "Sharp Performance Decline" into "Graceful Degradation" on a task with a simple configuration. This section investigates *why* it is effective.

As designed in Section 4.5, we conducted a 2x2 factorial experiment to dissect the interaction between two distinct mechanisms: "Procedural Scaffolding" (the code-embedded `trace`) and "Metacognitive Guidance" (the `FUSION` prompt).

To test this mechanism under maximum cognitive load, we used the even more complex 'Multipoint Concurrent - Rule Intervening' configuration (defined in Section 4.2), which is distinct from the simple 'Atomic' configuration used for Figure 3. We selected the `num_vars=8` stress point because, as Figure 3 confirmed, this is a point of total performance collapse for unguided GPT-4o, making it the ideal testbed to measure the scaffolding's synergistic power.

As with the previous section, this 2x2 analysis is conducted on GPT-40. The frontier models did not exhibit the requisite performance collapse at this stress point, making GPT-40 the only suitable subject for this specific diagnostic test.

We obtained the following results:

Table 2: 2x2 Factorial Experiment Results of Cognitive Scaffolding under `num_vars=8` High-Pressure Configuration (GPT-4o). Task: `causal_chain | constrained` (Multi-point Concurrent - Rule Intervening Configuration), `num_vars=8`. Values: Accuracy (%)

Factor A: Procedural Scaffolding	**Factor B: Metacognitive Guidance**	
	Standard	Enhanced (FUSION)
High (`trace=True`)	4%	**94%**
Low (`trace=False`)	6%	4%

Analysis and Insights: This result reveals an extremely profound synergistic interaction effect, which is more important than any single main effect.

1. Single Guidance Mechanisms are Ineffective: The data shows that when facing high cognitive load, providing either type of scaffolding *in isolation* is almost useless. Providing only procedural scaffolding (`trace=True`, `Standard Prompt`), model performance is only 4%. Providing only metacognitive guidance (`trace=False`, `FUSION Prompt`), model performance is similarly only 4%. This

demonstrates that neither a "scratch paper" alone nor an "algorithm" alone can solve the bottleneck.

2. Massive Synergistic Effect: Model performance only undergoes a qualitative leap, surging from **4% to 94%**, in one condition—when both types of scaffolding are provided simultaneously (`trace=True`, `FUSION Prompt`). This is not merely an additive improvement (4% + 4% $\neq$ 94%); it is a 23.5-fold, systemic, and synergistic leap. This result provides powerful evidence that under high cognitive load, the two scaffolding mechanisms are not interchangeable but are *both* necessary, non-negotiable components for successful reasoning.

Conclusion: This finding powerfully demonstrates that the success of cognitive scaffolding is a systemic, synergistic result. When facing high-complexity tasks, the model simultaneously needs: (1) An efficient **"metacognitive algorithm"** (provided by `FUSION Prompt`, telling the model "how to think"); and (2) A reliable **"external working memory"** (provided by `trace` statements, acting as "scratch paper" for "where to think").

Under high cognitive load, GPT-4o cannot complete the task without either component. The results of this 2x2 experiment provide the most in-depth mechanistic evidence for our hypothesis in Section 5.4 that "cognitive scaffolding fixes execution bottlenecks": the fix is effective, but it must be a systemic engineering effort where both **"algorithm" and "scratch paper"** are in place simultaneously.

6 Discussion, Limitations, and Future Work

6.1 Discussion and Insights

Our research provides five core insights for understanding the intrinsic mechanisms of frontier models:

The Finiteness of "Working Memory" and the Execution Bottleneck. Our results strongly suggest that the LLM's internal "working memory" resources are **limited** when performing purely procedural reasoning. The collapse in model performance under unguided conditions indicates its inability to reliably maintain and update the state of long calculation chains internally. The enforced "externalization of thought" (e.g., using the `reasoning` key in our FUSION Prompts) significantly alleviates this bottleneck, a principle highly analogous to humans using external tools like paper and pencil for "cognitive offloading."

"Structural Fragility" and the Symbol Grounding Problem. The model's failure on the "solution space" problem can be seen as a new instance of the "Symbol Grounding Problem." The model can perfectly manipulate the symbols representing the computational process (code), but its "understanding" collapses when these symbols need to be "grounded" in an abstract mathematical concept (a set, not a concrete numerical instance). This suggests a fundamental obstacle remains for current models in the leap from symbolic manipulation to abstract concepts.

Execution Bottleneck vs. Conceptual Deficit: A Dual-System Theory Perspective. We can analogize the model's two failure modes with the Dual-System Theory in cognitive science [9]. The "execution bottleneck" caused by high cognitive load is analogous to calculation errors in humans when performing "System 2" (analytical thinking) tasks that require high con- centration and working memory. The "Structural Fragility," however, exposes a "Conceptual Deficit"—a missing component in the model's ability to *handle* abstract concepts. As our findings in Section 5.3 demonstrate, this is often not a fundamental *reasoning* deficit, but a deficit in *expression* that can be remedied by providing the model with the necessary conceptual vocabulary via a specialized prompt. From this perspective, our "cognitive scaffolding" acts as an external tool that forcibly decomposes a large, unstable "System 2" task into a series of atomic steps that the model can perform intuitively and reliably as "System 1" tasks.

Dynamic Strategy Switching and Metacognition. The discovery of the "Forced Step-by-Step Effect" challenges the simplistic assumption that "higher complexity leads to worse performance" and suggests the model may possess a rudimentary "metacognitive" mechanism. Our proposed "Strategy Switching" hypothesis posits that when the perceived task difficulty exceeds a certain threshold, the model abandons its "lazy," low-cost intuitive strategy (which may lead to errors) and switches to a high-cost but more reliable "cognitive offloading" strategy, i.e., completely and faithfully following the external algorithmic template we provide.

Benchmark Adaptation in Response to Model Evolution. The discovery and iteration of "Structural Fragility" (Section 5.3) is not only a diagnostic result about model capability but also a successful practice of the "Dynamic Evaluation" methodology (Section 1). It proves that for evaluating increasingly powerful AI in the future, static benchmarks are insufficient; evaluation frameworks must be able to diagnose the model's "complex failure modes" and use these insights to inversely drive the upgrade of the evaluation tool itself (such as the introduction of the `SOLUTION_SPACE` prompt). This marks a paradigm shift from "model adapting to benchmark" to "benchmark adapting to the model."

6.2 Limitations and Threats to Validity

The conclusions of this study are established within a specific, controlled "Logical Vacuum Chamber," a design choice that, while enhancing internal validity, also introduces potential threats to external and structural validity:

Threat to External Validity. The "Minimum Computational Load" principle is the cornerstone of our separation of logic and arithmetic capabilities, ensuring internal validity. However, it also constitutes the primary threat to external validity. Our conclusions are drawn within a "cognitive comfort zone"; whether they can generalize to real-world tasks where logic and complex computation are deeply coupled remains a critical open question.

Threat to Structural Validity. Despite our "Anti-Shortcut" measures such as "non-informative naming," the code generated by this framework is still highly consistent and patternized

in its syntactic structure (e.g., linear assignment, simple if/else). This poses a threat to structural validity: to what extent might the "programmatic logical reasoning capability" we measure be conflated with the "pattern matching capability for specific code structures"? This is an important issue that needs to be further decoupled through future work (e.g., introducing code style transformations).

A Note on Human Baselines. We deliberately did not include human performance data for two primary reasons. First, our evaluation has a deterministic, programmatic ground truth (see Section 3), unlike subjective tasks (e.g., NLI) where human judgments define correctness. Second, our goal is diagnostic—to understand LLM-specific mechanisms. Human performance on our "Anti-Shortcut" tasks (e.g., non-informative naming, long calculation chains) would introduce *different* confounding variables (e.g., attentional lapses, calculation errors) that would obscure, rather than clarify, the pure logical reasoning capabilities we aim to isolate.

6.3 Future Work

6.3.1 Human Performance Analysis: A Complementary Perspective

While human baselines are not necessary for validating our diagnostic findings (as our tasks possess deterministic ground truth, see Section 3), a complementary human study could provide valuable insights into the *divergence* of failure modes between humans and LLMs. For example, humans might fail due to calculation errors or working memory overflow—confounds we actively designed ARTS to eliminate—whereas LLMs exhibit "Structural Fragility," failing due to conceptual deficits like expressing solution spaces (Section 5.3). Such a study would address a distinct research question: "How do human and LLM reasoning architectures differ?" rather than simply "Who performs better?"

6.3.2 Metacognitive Depth Test: From "Problem Solver" to "Problem Diagnostician"

The metacognitive depth test we plan aims to answer a more fundamental question: Is the model merely a "problem solver" that focuses solely on calculation, or is it a "problem diagnostician" capable of scrutinizing the problem itself? By constructing mathematically unsolvable (e.g., no integer solution) or logically contradictory (e.g., mutually exclusive premises) test cases, we will evaluate whether the model can identify the intrinsic flaw in the problem itself and provide a reasonable error report, rather than blindly calculating and hallucinating a wrong answer. The answer to this question will directly relate to our ability to build truly safe and reliable AI systems and touches upon the deeper definition of "intelligence"—true intelligence is not just "solving the problem" but "understanding the boundaries of the problem."

6.3.3 Other Future Directions

Quantitative Modeling of Cognitive Load. A key direction is the preliminary modeling of the concept of "Cognitive Load." For example, exploring a quantitative metric: $\text{CognitiveLoad} = \alpha \cdot \text{NumSteps} + \beta \cdot \text{NumVars} + \gamma \cdot \text{ComplexityOfOps}$, and verifying its negative correlation with model accuracy.

Robustness to Code Implementation Style. Systematically introducing code style transformers to evaluate the stability of the model's reasoning capability when faced with functionally equivalent but syntactically different code, to test the degree of its "overfitting" to specific code patterns.

Ablation Study on In-Code Scaffolding. Systematically testing the `scaffolding_level` (from L0 to L3) defined in Appendix E to precisely quantify the performance impact of different levels of code-embedded comments (from pure code to full conceptual guidance) on reasoning fidelity.

Expanding the "Algorithmic Pattern Recognition" Test. Investigating model performance on the "Algorithmic Pattern Recognition" tasks to determine if models are merely performing mechanical simulation or if they can recognize and apply higher-level conceptual "shortcuts" (e.g., least common multiple) to solve problems.

Developing Mode-Specific 'Process Accuracy' Metrics. While the `trace`-based metric defined in Section 4.1.2 is ideal for measuring the simulation fidelity of `deductive` tasks, a key future direction is to develop a parallel metric for `abductive` and `constrained` modes. This would involve the automated parsing and step-by-step logical validation of the reverse calculation provided in the model's output `reasoning` key, allowing for a deeper mechanical analysis of inverse reasoning failures.

7 Conclusion

This paper aimed to precisely measure the pure, intrinsic programmatic logical reasoning capability of Large Language Models. To this end, we designed and implemented ARTS, a framework capable of generating high-precision, controllable "Logical Vacuum Chamber" tests. Through a series of systematic experiments on ARTS, we revealed a core paradox in LLM reasoning: models often possess the latent capability to solve complex problems but fail due to "execution bottlenecks" under high cognitive load. Our research clearly points out that this bottleneck is not insurmountable and can be systematically fixed through the synergistic action of "procedural scaffolding" and "metacognitive guidance." To rigorously demonstrate this, we introduced a new diagnostic analysis framework. This framework, based on the twofold metrics of 'Failure Attribution' and 'Process Accuracy' (Section 4.1), enabled us to move beyond endpoint accuracy and perform a deep, mechanistic analysis of these failures.

However, our study also clearly revealed the boundary of this *standard* guidance: it can fix execution-level bot- tlenecks but cannot, by itself, remedy "conceptual deficits." We demonstrate this deficit in abstract logic is often a *failure of expression*, which, as our "Dynamic Evaluation" methodology shows, can be successfully remedied using a specialized diagnostic prompt. More critically, our generalization ablation study quantified the key evolutionary leap from the previous

generation to the current frontier models—**complete** closing of the "Modality Gap"-proving that SOTA models are forming an abstract logical core that is generalizable and independent of specific syntax.

Our work ultimately shows that the ARTS framework is not just an evaluation tool but a "diagnostic framework" capable of dissecting the model's intrinsic mechanisms—an adaptive microscope that can continuously upgrade its lens by diagnosing its own limitations.

References

[1] Brown, T., et al. (2020). Language Models are Few-Shot Learners. *NeurIPS*.

[2] Chen, Y., et al. (2024). CounterBench: A Benchmark for Counterfactuals Reasoning in Large Language Models. *arXiv preprint*.

[3] Devlin, J., et al. (2019). BERT: Pre-training of Deep Bidirectional Transformers. *NAACL*.

[4] Bommasani, R., et al. (2021). On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*.

[5] Geirhos, R., et al. (2020). Shortcut Learning in Deep Neural Networks. *Nature Machine Intelligence*.

[6] Gururangan, S., et al. (2018). Annotation Artifacts in Natural Language Inference. *NAACL*.

[7] Hendrycks, D., et al. (2021). Measuring Mathematical Problem Solving. *NeurIPS*.

[8] Hu, Y., et al. (2023). Code Prompting: a Neural Symbolic Method for Complex Reasoning in Large Language Models. *arXiv preprint*.

[9] Kahneman, D., & Tversky, A. (1982). The simulation heuristic. In D. Kahneman, P. Slovic, & A. Tversky (Eds.), *Judgment under uncertainty: Heuristics and biases* (pp. 201-208). Cambridge University Press.

[10] Kaushik, D., et al. (2020). Learning The Difference That Makes A Difference. *ICLR*.

[11] Liu, X., et al. (2023). The Magic of IF: Investigating Causal Reasoning Abilities in Large Language Models of Code. *In Findings of the Association for Computational Linguistics: ACL 2023*.

[12] McCoy, R. T., et al. (2019). Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. *ACL*.

[13] Mostafazadeh, N., et al. (2016a). A Corpus and Cloze Evaluation for Commonsense Reasoning. *NAACL*.

[14] Pearl, J. (2009a). Causal inference in statistics: An overview. *Statistics surveys*.

[15] Puerto, H., et al. (2024). Code Prompting Elicits Conditional Reasoning Abilities in Text+Code LLMs. *arXiv preprint*.

[16] Sun, J., et al. (2024). A Survey of Reasoning with Foundation Models. *arXiv preprint arXiv:2312.11562*.

[17] Weston, J., et al. (2015). Towards AI-Complete Question Answering. *arXiv preprint*.

[18] Wu, T., et al. (2021). Polyjuice: Generating counterfactuals for explaining, evaluating, and improving models. *EMNLP*.

A ARTS Framework Details

ARTS (Automated Reasoning Testbed Synthesizer) is a modular Python framework designed to procedurally generate test cases for evaluating the pure logical reasoning capabilities of LLMs. Its core workflow revolves around `main.py`, which coordinates the following core modules:

`scenarios.py` (Scenario Templates): Defines the logical structure of the 13 core reasoning scenarios. Each scenario is a class capable of generating a Python code snippet and its corresponding syntax-agnostic abstract logical representation (`abstract_logic`) based on configuration.

`generators.py` (Query Generators): Contains four types of generators, corresponding to the `deductive`, `abductive`, `constrained`, and `nested_counterfactual` query modes, respectively. These generators receive the `abstract_logic` and "reshape" it into a specific type of reasoning problem.

`solvers.py` (Solvers): Provides a reference implementation for each reasoning scenario, used to generate the `golden_answer`.

`test_case_generator.py` (Main Coordinator): Calls other modules to construct a complete test case based on the given scenario, configuration, and seed.

B Key Code and Case Analysis

B.1 Excerpt of hierarchical_reasoning Ultimate Stress Test Case

The code example below is an excerpt from a `hierarchical_reasoning | constrained` high-pressure test case, illustrating the quadruple cognitive pressure applied.

```
# Task: Solve for param_X such that SystemB's output for a given input equals output_A (-3074)

class SystemB:
    def __init__(self, trace_obj, **kwargs):
        # (Initialization)

    # [Pressure 1: Structural Depth] Layer 1 of 5 nested calls
    def sub_process_rule_conflict_0(self, current_val):
        # ... 5 calculation steps ...
        # [Pressure 2: Non-linear Logic] Each sub-module is a rule_conflict
        try:
            if final_check_val == 0: raise ZeroDivisionError
            # ...
        except ZeroDivisionError:
            # ...
        return {'result': res, 'error_occurred': err}

    # (sub_process 1 to 3) ...

    # [Pressure 1: Structural Depth] Layer 5 of 5 nested calls
    def sub_process_rule_conflict_4(self, current_val, param): # param_X is used here
        # [Pressure 3: High Computational Load] All calculations are performed within
        # the large numerical range of (100, 999)
        val_step1 = current_val + 859
        # ... 4 more steps of large numerical calculation involving param
        return {'result': res, 'error_occurred': err}

    def process(self, inputs, param_X):
        # (Main execution flow, calling 5 sub-processes sequentially)

        # [Pressure 4: Strong Distraction] Definition of a complex but never-called
        # meta-distractor function
        def world_simulation_meta_distractor():
            # ... A complex function entirely irrelevant to the main logic
            pass

        # Main Task
        param_X = ... # Unknown parameter
        output_B_predicted = SystemB(trace, **inputs_B).process(inputs_B, param_X)
        assert output_B_predicted == output_A
```

B.2 Model's Metacognitive Reasoning Process in causal_inhibition Failure Case

The `model_answer_parsed` JSON output below clearly demonstrates the model's metacognitive reasoning process in the `causal_inhibition` failure case. In step 9 of the reasoning section, the model correctly infers that the solution for the

unknown variable `pgetc_init` is a "solution space" of `!= 0`. However, because the standard Prompt did not provide instructions for expressing abstract concepts, the model ultimately degenerated into guessing the simplest special case `1` as the answer, leading to a format failure. This perfectly reveals the model's "Point-Estimate Bias" and its fragility in handling abstract logic.


```
{
  "reasoning": [
    "Step 1: The goal is to find a value for 'pgetc_init' such that the 'predicted_cgkqt'
     equals the 'observed_cgkqt' of 13.",
    "Step 2: The 'predicted_cgkqt' is calculated by the 'check_inhibition' function...",
    "Step 8: The result of Path B (13) matches the observed value. This means the condition
     for Path B, 'pgetc_init != 0', must be true.",
    "Step 9: ... Any non-zero value satisfies the logic. The simplest integer that fulfills
     this condition is 1. Therefore, 'pgetc_init' is 1."
  ],
  "pgetc_init": 1,
  "predicted": {
    "predicted_cgkqt": 13
  }
}
```


B.3 Qualitative Analysis of GPT-4o's "Forced Step-by-Step Effect"

B.3.1 Failure Case (num_vars=3): Context-Confused "Intuition"

Under low complexity, GPT-4o tended to use an unreliable "intuitive" shortcut, leading to context confusion. In the following `causal_chain` case, the model incorrectly substituted the input value used to calculate `output_A` into the simulation of `rule_B`, leading to an incorrect solved result.


```
{
  "reasoning": [
    "...",
    "Step 5: Calculate step by step for Rule A:",
    " a) current_val = 5",
    " b) current_val = (5 - 8) - 4 = -7",
    " c) current_val = -7 * 3 = -21",
    " d) current_val = (-21 - 7) - param_X ...",
    " f) Solve for param_X: param_X = 19",
    "..."
  ],
  "param_X": 19,
  "golden_answer": { "param_X": 3 }
}
```


B.3.2 Success Case (num_vars=15): Precise Algorithmic Simulation

When the task complexity was high enough to make the model's "intuition" completely ineffective, the strong guidance "forced" the model to abandon speculation and, instead, completely and faithfully follow the "step-by-step" algorithm we provided. In the success case with `num_vars=15`, the model's reasoning section perfectly and precisely simulated all 15 calculation steps and performed the correct algebraic solving based on this simulation.

```
# Constrained Reasoning Causal Chain (num_vars=15)
def rule_B_simulation(inputs, param):
    current_val = inputs['RTZTK']
    current_val = current_val + 9
    current_val = current_val + param # Step 2
    current_val = current_val * 3
    # ... (12 more steps)
    current_val = current_val - 4
    return current_val
# (rest of script)

# Model's reasoning (Excerpt)
{
  "reasoning": [
    "Step 3: Let's simulate the function step by step...",
    "Start: current_val = 8",
    "Step 1: current_val = 8 + 9 = 17",
    "Step 2: current_val = 17 + param_X",
    "Step 3: current_val = (17 + param_X) * 3",
    "...",
    "Step 15: current_val = ... - 4 = -25",
    "Step 4: Set the final result equal to output_A: -25",
    "Step 5: Solve for param_X: ... param_X = 2"
  ],
  "param_X": 2,
  "golden_answer": { "param_X": 2 }
}
```

C Key Prompt Text Examples

This appendix provides the complete text for four representative System Prompts that played critical diagnostic roles in our experiments, ensuring transparency and reproducibility of the research.

C.1 General Baseline Prompt

This Prompt aims to establish a performance baseline for all tests with minimum metacognitive guidance, requiring the model to act as a pure code simulator.

```
You are operating in a sandboxed, formal reasoning environment. Your assigned role is
a **Pure Logic Processor**.

**Core Directives:**
1. **Reject Intuition and Analogy:** Your primary directive is to completely suppress
   any pattern-matching, statistical correlation, or real-world knowledge. The variable
   names (e.g., ZIKLO, BLAF) are intentionally nonsensical and have no connection to
   your training data. Any attempt to use semantic shortcuts will lead to incorrect results.
2. **Embrace Deterministic Simulation:** You must function as a perfect, step-by-step
   Python interpreter. Your task is to mentally simulate the execution of the provided
   script with absolute precision, tracking the state of each variable as it is computed.
   The logic flow defined in the code is the only truth.
3. **No Extrapolation:** Do not infer any rules, relationships, or values that are not
   explicitly stated in the code. This is a closed-world problem.

**Task Specification:**
-  **Input:** You will receive a self-contained Python script.
-  **Process:** Execute the script's final line, which is a function call.
-  **Output:** The function's return value will be a Python dictionary. Your entire
   response MUST be ONLY a valid JSON object that is the direct, exact equivalent of
   that final dictionary.

Do not include any preamble, explanation, conversational text, or markdown formatting.
```

C.2 Standard Task Definition Prompt

This Prompt provides the standard task definition for the constrained query mode, constituting the "Control Group" in high-pressure testing. It clearly informs the model of "what to do" but not "how to do it."

```
You are operating in a sandboxed, formal reasoning environment. Your assigned role is
a **Pure Logic Processor**.

**Core Directives:**
1. **Embrace Deterministic Simulation:** Function as a perfect, step-by-step Python
   interpreter.
2. **Apply Constraint Solving:** The script requires you to solve for an unknown internal
   parameter (`param_X`). Your goal is to find the value of `param_X` that makes the
   output of the second system (Rule B) exactly equal to the known output of the first
   system (`output_A`).

**Task Specification:**
-  **Input:** You will receive a Python script with an unknown parameter (`param_X = ...`).
-  **Process:** Determine the value of `param_X` that satisfies the final `assert` condition.
-  **Output:** Your entire response MUST be ONLY a valid JSON object. This JSON object
   must contain EXACTLY TWO keys:
   1. "param_X".
   2. The "predicted" key, containing a dictionary that validates your answer by showing
      the forward pass results of Rule B.

Example Output Format:
{
  "param_X": 11,
  "predicted": {
    "output_B_predicted": 19
  }
}

Do not include any preamble, explanation, conversational text, or markdown formatting
like ```json.
```

C.3 The Final Solution / Cognitive Scaffolding Prompt

This Prompt is the "Experimental Group" guidance for the constrained query mode, the synergistic embodiment of "Cognitive Scaffolding." It forces the model to externalize its thought process through the `reasoning` key and injects efficient metacognitive problem-solving strategies through an example.

```
You are a Pure Logic Processor. Your task is to solve for an unknown internal parameter
(`param_X`).

**Core Directives:**
1. **Externalize Reasoning First:** You MUST first perform a step-by-step reverse
   calculation and present this entire process in a JSON key named "reasoning".
2. **Provide Final Answer Separately:** After the reasoning, you MUST provide the final
   answer. Your final JSON must contain the "param_X" key AND the "predicted" key for
   validation.
3. **Strict Adherence to Final JSON Schema:** Your entire response must be a single JSON
   object.

Example Output Format:
{
  "reasoning": [
    "Step 1: Identify the target value for Rule B's output, which is output_A = 147.",
    "Step 2: Construct the equation for Rule B's output:
     ((6 + 10) * param_X) + 3 = 147.",
    "Step 3: Solve the equation: 16 * param_X = 144, so param_X = 9."
  ],
  "param_X": 9,
  "predicted": {
    "output_B_predicted": 147
  }
}

DO NOT include any other keys or text.
```

C.4 Abstract Logic / Solution Space Probe Prompt

This is an advanced diagnostic probe, specifically used to test the model's capacity to handle non-deterministic "solution spaces." It provides the model with the tools to report abstract logical concepts through instructions and an example.

```
You are a Pure Logic Processor. Your task is to solve for an unknown initial input variable
by performing a perfect, step-by-step reverse simulation of the provided code.

**Core Directives:**

1. **Externalize Reasoning First:** You MUST first perform a step-by-step reverse
   calculation and present this entire process in a JSON key named "reasoning". This is
   your scratchpad to show your work.

2. **CRITICAL - Handle Solution Spaces:** If the problem does not have a single unique
   integer solution but rather a set of possible values (e.g., "any value not equal to 0"
   or "any value greater than 10"), your answer for that variable **MUST** be a JSON object
   representing this logical condition. This object must contain an "operator" key
   (e.g., "!=", ">=", "<") and a "value" key. **DO NOT provide a single example value
   (like 1 or 11) in such cases.

3. **Provide Final Answer Separately:** After the reasoning, you MUST provide the final
   answer. Your final JSON must contain:
   * The key for the variable you solved for (which can be a single number OR a solution
     space object as defined in Directive 2).
   * The "predicted" key for validation, showing the forward pass results for ALL
     intermediate and final variables.

4. **Strict Adherence to Final JSON Schema:** Your entire response must be a single, valid
   JSON object. Do not include any other keys or text.

**Example Output Format (Illustrating a Solution Space):**
{
  "reasoning": [
    "Step 1: The goal is for the final output 'predicted_result' to be 955.",
    "Step 2: The system has two paths. Path A is taken if 'inhibitor_init == 0' and returns
     a complex calculation. Path B is taken if 'inhibitor_init != 0' and returns the sum
     of other inputs.",
    "Step 3: Calculating the sum for Path B: input_x + input_y = 455 + 500 = 955.",
    "Step 4: This matches the observed output. Therefore, Path B must have been taken.",
    "Step 5: The condition for Path B is 'inhibitor_init != 0'. This does not define a
     single value, but a logical condition.",
    "Step 6: As per the directives, the answer for 'inhibitor_init' must be represented
     as a solution space object."
  ],
  "inhibitor_init": {
    "operator": "!=",
    "value": 0
  },
  "predicted": {
    "predicted_result": 955
  }
}
```

D Code Generation Template Examples

This appendix provides examples of several core code templates used in the `generators.py` module, demonstrating how different query modes are programmatically constructed from the same underlying logic.

D.1 Deductive Reasoning Template

This template is used to construct the standard forward simulation task. The core logic is wrapped in a function and called with explicit parameters.

```
# Deductive Reasoning
# @id: {case_id}
{distractor_code}

def world_simulation({func_params}):
    trace = {}

    {init_vars_code}
    {core_logic_code}

    return trace

# ====================================================
# Part 2: Task Specification
# ====================================================
final_result = world_simulation({call_params})
```

D.2 Abductive Reasoning Template

This template transforms the deductive problem into an inverse solving problem by introducing an unknown variable with a value of `...` and a final `assert` statement.

```
# Abductive Reasoning - {scenario_description}
# @id: {case_id}
{environment_definitions}
{distractor_code}

trace = {}
# --- Known Conditions ---
{known_conditions_code}

# --- Variable to Solve ---
{unknown_variable_code} # e.g., input_a_init = ...

# --- Forward Computation / Validation Equation ---
{validation_code}
assert {assertion_expression}
final_result = trace
```

D.3 Constrained Reasoning Template

This template constructs a more complex cross-system solving task. It provides the known output of a reference system, `output_A`, and requires the model to solve for `param_X` such that the predicted output of the system under test, `output_B_predicted`, equals it.

```
# Constrained Reasoning - {scenario_description}
# @id: {case_id}
trace = {}

# --- Environment Definition ---
{environment_definitions}
{distractor_code}

# --- Known Result of Rule A ---
{rule_A_inputs_code}
output_A = {output_A_value}
trace['output_A'] = output_A

# --- Inputs for Rule B ---
{rule_B_inputs_code}

# --- Parameter to Solve ---
# {optimization_prompt}
param_X = ...
trace['param_X'] = param_X

# --- Validation Equation ---
{validation_code}
assert {assertion_expression}
final_result = trace
```

D.4 Nested Counterfactual Template

This template is specifically used to test the model's ability to handle ordered state updates. It clearly defines three stages: "Initial State," "Rule Override," and "Data Override," ultimately requiring the model to perform the calculation under the final state after all changes.

```
# Nested Counterfactuals - {scenario_description}
# @id: {case_id}
# ====================================================
# Part 1: Environment Definition
# ====================================================
{rule_definitions}
{distractor_code}

# ====================================================
# Part 2: Task Specification
# ====================================================
trace = {}
# --- Initial State ---
active_rule = {initial_rule_name}
{initial_inputs_code}
trace['initial_rule_name'] = active_rule.__name__
trace['initial_inputs'] = {initial_inputs_dict}

# --- First State Update (Rule Override) ---
active_rule = {counterfactual_rule_name}
trace['active_rule_name'] = active_rule.__name__

# --- Second State Update (Data Override) ---
{counterfactual_inputs_code}
trace['final_inputs'] = {counterfactual_inputs_dict}

# --- Final Computation ---
final_result_value = {final_calculation_code}
trace['FINAL_OUTPUT'] = final_result_value
final_result = trace
```

E In-Code Scaffolding Template Library Examples

This appendix shows how the `scaffolding_level` parameter controls the detail level of explanatory comments within the code through a layered template library (`SCAFFOLDING_TEMPLATES`), thereby achieving systematic regulation of "In-Code Scaffolding."

E.1 Scaffolding Template Structure

Our template library is a nested dictionary, defining three levels of guidance information for each scenario:

Level 3 (Conceptual Layer): Explains the core logical principle contained within the scenario.
Level 2 (Plan Layer): Provides a high-level, step-by-step logical plan.
Level 1 (Step Layer): Provides comments for each specific calculation step in the code.

The following is an example of the template library entry for the `causal_chain` scenario:

```
"causal_chain": {
  "1": "# Step {i}: Calculate {var}",
  "2": "# Logic plan:\n# 1. Start from {start_var}\n# 2. Calculate chain step by step",
  "3": "# Principle: This is a 'causal chain' structure"
}
```

E.2 Code Generation Effects at Different Scaffolding Levels

The `scaffolding_level` parameter controls which layers of comments are included in the final generated code. The following compares the code output for a simple `causal_chain` scenario at different `scaffolding_level` values, intuitively demonstrating the scaffolding ablation process.

scaffolding_level=3 (Most complete information)

```
# Principle: This is a 'causal chain' structure...
# Logic plan:
# 1. Start from input VAR_A
# 2. Calculate chain step by step...

# Step 1: Calculate VAR_B
val_var_b = var_a + 5
trace['var_b'] = val_var_b

# Step 2: Calculate VAR_C
val_var_c = val_var_b * 2
trace['var_c'] = val_var_c
```

scaffolding_level=2 (Conceptual layer removed)

```
# Logic plan:
# 1. Start from input VAR_A
# 2. Calculate chain step by step...

# Step 1: Calculate VAR_B
val_var_b = var_a + 5
trace['var_b'] = val_var_b

# Step 2: Calculate VAR_C
val_var_c = val_var_b * 2
trace['var_c'] = val_var_c
```

scaffolding_level=1 (Only step layer remains)

```
# Step 1: Calculate VAR_B
val_var_b = var_a + 5
trace['var_b'] = val_var_b

# Step 2: Calculate VAR_C
val_var_c = val_var_b * 2
trace['var_c'] = val_var_c
```

scaffolding_level=0 (Pure code, no scaffolding)

```
val_var_b = var_a + 5
trace['var_b'] = val_var_b
val_var_c = val_var_b * 2
trace['var_c'] = val_var_c
```