From Laboratory to Real-World Applications: Benchmarking Agentic Code Reasoning at the Repository Level

Jia Li
The Chinese University of Hong Kong
Hong Kong, Hong Kong
linsayli@link.cuhk.edu.hk

Yuxin Su*
Sun Yat-sen University
Zhu Hai, China
suyx35@mail.sysu.edu.cn

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, Hong Kong
lyu@cse.cuhk.edu.hk

Abstract

As large language models (LLMs) evolve into autonomous agents, evaluating *Repository-Level Reasoning*, the ability to maintain logical consistency across massive, interdependent file systems, has become critical. Current benchmarks typically fluctuate between isolated code snippets and black-box evaluations. We present REPOREASON, a white-box diagnostic benchmark centered on *Abductive Assertion Verification*. To eliminate memorization while preserving authentic logical depth, we implement an *Execution-Driven Mutation* framework that utilizes the environment as a *Semantic Oracle* to regenerate ground-truth states. Furthermore, we establish a fine-grained diagnostic system using dynamic program slicing, quantifying reasoning via three orthogonal metrics: *ESV* (Reading Load), *MCL* (Simulation Depth), and *DFI* (Integration Width). Comprehensive evaluations of frontier models (e.g., Claude-4.5-Sonnet, DeepSeek-v3.1-Terminus) reveal a prevalent *Aggregation Deficit*, where integration width serves as the primary cognitive bottleneck. Our findings provide granular white-box insights for optimizing the next generation of agentic software engineering.

1 Introduction

The paradigm of software engineering is undergoing a fundamental shift: from human-centric coding to agentic autonomous development (Jiang et al., 2025; Hou et al., 2024). As Large Language Models (LLMs) evolve into autonomous software agents, the core competency required has escalated from performing localized execution reasoning within isolated code snippets—akin to controlled laboratory exercises—to maintaining logical consistency across extensive, interdependent file systems—a capability we define as Repository-Level Reasoning. This capability imposes profound challenges on LLMs, demanding not only code synthesis but also the agency to explore vast contexts to pinpoint target definitions, aggregate multi-source information across complex dependencies, and perform long-chain reasoning to accurately track and compute state mutations.

However, a critical gap exists between this growing demand and current evaluation methods. Existing benchmarks largely fall into two extremes. On one side, function-level reasoning datasets, such as CRUXEval (Gu et al., 2024) and REval (Chen et al., 2024), study code execution reasoning within isolated snippets. While these provide analytical rigor, they operate in a laboratory setting that lacks the cross-file dependencies and complex architectural contexts found in real-world software engineering. On the other side, frameworks like SWE-bench (Jimenez et al., 2024) evaluate agents in wild repository environments but function as black-boxes: they check whether a problem is solved but offer little insight into why an agent fails. Without detailed diagnosis, the path to improving agent reasoning remains unclear.

To address this gap, we introduce RepoReason, a repository-level code reasoning benchmark designed as a white-box diagnostic tool. Unlike traditional generation tasks, RepoReason shifts to a verification-centric approach: "Given the complex execution history of this repository, what is the current system state?" By requiring models to derive deterministic values that satisfy assertions rather than writing implementation logic, we separate core logical reasoning from syntactic noise (*e.g.*, formatting errors, indentation slips, or API name hallucinations). This ensures that every fail-

*Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 8845–8869

ure reflects a genuine deficit in reasoning rather than superficial coding errors.

To ensure the benchmark reflects real-world complexity, we extract task instances from a curated set of mainstream Python repositories (*e.g.*, `toolz` (Developers, 2025b), `sympy` (Team, 2025c), `jinja2` (Pallets, 2025)), challenging models with authentic, deep logic. Beyond static code extraction, we perform concrete execution of the repositories' built-in unit test suites to capture granular runtime traces. By designing structural metrics based on these traces—such as stack depth, function call density, and cross-file dependency breadth—we quantitatively filter and retain only the most complex reasoning tasks.

Constructing such a benchmark requires navigating between two distinct failure modes of current evaluations: Spurious Correlation (guessing via shallow patterns) and Rote Memorization (recalling via pre-training data). To address this, we propose a two-stage defense mechanism. First, to prevent shortcut learning via visual pattern matching, we reject raw trace values and instead identify Unit Test Assertions as our semantic anchors. While raw traces are often localized and easily inferred from immediate code patterns, assertions function as semantic milestones that encapsulate high-level developer intent, strategically positioned to validate the system state after complex logical operations. They force Abductive Reasoning (Josephson and Josephson, 1994), requiring the model to reconstruct the preceding execution history across complex call stacks to deduce the verified state.

However, a significant obstacle remains: many assertions within these mainstream repositories' unit tests have already been encountered by LLMs during pre-training. This leads to a risk of data leakage (Deng et al., 2024). To resolve this, we implement an Execution-Driven Mutation methodology. We treat the code execution environment as a Semantic Oracle. By keeping the reasoning logic (the call graph filtered via structural metrics) intact but perturbing the program inputs, we effectively sever the model's memory retrieval path. We then re-execute the entire repository context to capture the new ground-truth state, subsequently masking these verified values to create cloze-style reasoning tasks. Finally, to ensure these captured states are objectively evaluable, we introduce a Deterministic Value Protocol, which filters runtime data to eliminate representational drift and ensure stable, unique ground-truth values. This design ensures that while the reasoning logic remains authentic and complex, the specific state values are unmemorized and deterministic.

Finally, through an extensive evaluation of a diverse range of frontier models including Claude-Sonnet-4.5 and DeepSeek-v3.1-Terminus, we demonstrate that even top-tier agents face severe performance degradation when confronted with high-entropy logic. Our results unveil a critical Aggregation Deficit where accuracy drops sharply as integration width increases, as well as a Cliff Effect in reading comprehension beyond 600 lines and a significant loss of consistency beyond 100 execution steps. These insights highlight that next-generation agents must transcend local reasoning and focus on high-breadth information synthesis and long-chain state consistency.

In summary, this paper makes the following contributions:

- **From Prediction to Abductive Verification:** We pioneer a repository-level reasoning paradigm that pivots from direct outcome prediction to abductive assertion verification. By centering tasks on masked assertions within interdependent file systems, we successfully decouple core architectural intelligence from surface-level syntactic noise.
- **The Semantic Oracle Framework:** We resolve the Richness-Contamination Dilemma through an Execution-Driven Mutation framework. By utilizing the execution environment as a Semantic Oracle, we sever data leakage paths via input perturbation and real-time state regeneration, ensuring tasks remain logically authentic yet unmemorized.
- **White-box Diagnostic Instrument:** Beyond end-to-end scoring, we introduce a granular diagnostic system powered by dynamic program slicing. We formalize three orthogonal cognitive metrics (Reading Load, ESV; Simulation Depth, MCL; Integration Width, DFI) that map reasoning failures to specific cognitive bottlenecks: context overload, state tracking deficits, and aggregation barriers.
- **Empirical Unveiling of Aggregation Deficits:** Through extensive evaluation of frontier models, we provide the first quantitative analysis of agentic reasoning trajectories across massive codebases (up to 776k+ LoC). Our findings identify DFI as the primary bottleneck for next-generation agents, empirically confirming

a severe aggregation deficit in information synthesis.

2 Related Work

2.1 Code Generation

Existing code benchmarks have evolved from function-level synthesis (*e.g.*, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021)) to class-level and repository-level completion (*e.g.*, ClassEval (Du et al., 2023), RepoBench (Liu et al., 2024)). While these frameworks increasingly reflect real-world complexity, they primarily target syntactic completion—the ability to produce correct code text. In contrast, RepoReason shifts the focus to semantic reasoning. By requiring the deduction of deterministic system states via assertion verification, it isolates core logical reasoning from the syntactic noise and hallucination risks inherent in code generation.

2.2 Code Reasoning

Recent benchmarks like CRUXEval (Gu et al., 2024) and REval (Chen et al., 2024) pioneered execution reasoning but are largely confined to isolated, lab-scale code snippets. While tools like CORE (Xie et al., 2025) incorporate static analysis tasks to evaluate reasoning, they lack the structural depth and code breadth (*e.g.*, deep inheritance, cross-module imports) characteristic of large-scale projects. RepoReason bridges this gap by applying deep execution simulation to massive repository scopes, challenging models to maintain logical consistency across extensive file systems.

2.3 Agentic Software Engineering

At the highest level, SWE-bench (Jimenez et al., 2024) evaluates agents on resolving GitHub issues. However, as an end-to-end evaluation, it acts as a black-box indicator of success or failure. RepoReason complements this by targeting the cognitive root causes of failure. By decoupling reasoning from code editing and utilizing fine-grained metrics (ESV, MCL, DFI), it provides the white-box diagnostic insights required to pinpoint an agent's specific bottlenecks in context handling or multi-source integration.

2.4 Summary Comparison

Table 1 provides a multi-dimensional comparison between RepoReason and mainstream code benchmarks, highlighting its unique advantages in diagnostic depth and structural complexity.

3 RepoReason: Design and Construction

In this section, we delineate the methodological framework of RepoReason, conceptualizing the construction pipeline as a rigorous five-phase defense system. Our objective is to transform authentic software logic into white-box diagnostic tasks while systematically neutralizing systemic vulnerabilities: spurious correlation, rote memorization, and logical superficiality. As illustrated in Figure 1, the pipeline orchestrates sequential gates: (1) Repository Curation to ensure environmental fidelity; (2) Structural Filtering via runtime traces to guarantee logical depth; (3) Execution-Driven Mutation to eliminate parametric leakage; (4) Task Instantiation governed by a deterministic value protocol for stability; and (5) Diagnostic Evaluation establishing fine-grained cognitive profiles. This integration ensures that each task instance necessitates authentic repository-level reasoning and robust state tracking, effectively precluding resolution via shallow pattern matching.

3.1 Design Philosophy: Assertions as Semantic Anchors

We explicitly reject two prevailing paradigms: traditional Input/Output (I/O) Prediction and Raw Trace Querying.

I/O Prediction Inefficiency: The traditional I/O Prediction paradigm (Gu et al., 2024; Chen et al., 2024), while effective for isolated algorithmic puzzles, is fundamentally ill-suited for repository-level reasoning. In authentic software ecosystems, runtime data structures, such as SymPy's symbolic expression trees or NetworkX's graph topologies, are often voluminous and complex, lacking the clear, concise boundaries found in synthetic tasks. Consequently, verifying the entire output state is inherently inefficient and noise-laden, as a significant portion of the data consists of dynamic metadata or irrelevant state rather than the target logic.

Spurious Correlation in Raw Traces: A straightforward approach might involve randomly querying variable values at specific locations (e.g., "What is the value of variable x at line 10?"). However, this paradigm of raw trace querying is fundamentally flawed for evaluating deep reasoning. It suffers from shortcut learning, as models can often retrieve answers through shallow static analysis of

Table 1: Comparison of **RepoReason** with existing code reasoning and understanding benchmarks.

Benchmark	Repo-Lvl	Scale (LoC)	Evaluation Paradigm	Metric Granularity	Diagnostic Depth
HumanEval (Chen et al., 2021)	×	∼10	Code Generation	Pass/Fail	Black-box
MBPP (Austin et al., 2021)	×	∼10	Code Generation	Pass/Fail	Black-box
ClassEval (Du et al., 2023)	×	∼40	Code Generation	Pass/Fail	Black-box
CRUXEval (Gu et al., 2024)	×	∼10	Execution Prediction	Pass/Fail	Black-box
REval (Chen et al., 2024)	×	∼10–40	Runtime Behavior	Pass/Fail	White-box (Local)
CORE (Xie et al., 2025)	×	21–100	Static Analysis	Pass/Fail	White-box (Static)
SWE-bench (Jimenez et al., 2024)	✓	1k – 1M+	Issue Resolution	Pass/Fail	Black-box
RepoReason (Ours)	✓	**1.2k – 775k+**	**Assertion Verification**	**Cognitive Metrics**	**White-box (Global)**

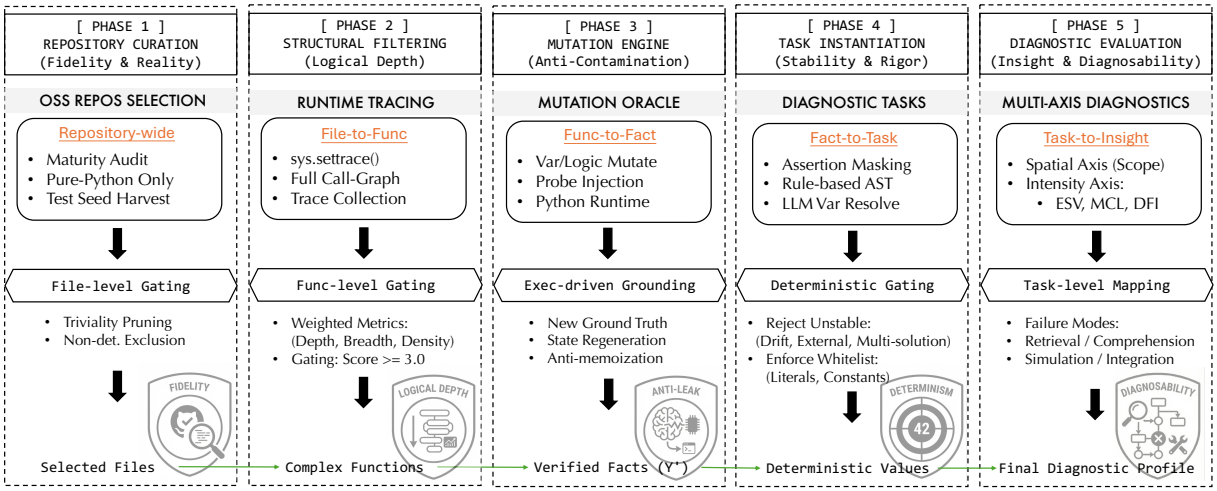

Figure 1: The overall architecture and multi-stage pipeline of RepoReason, encompassing repository curation, structural filtering, execution-driven mutation, task instantiation, and diagnostic evaluation.

nearby assignment statements, bypassing the need for genuine execution simulation.

The Solution: Assertions as Semantic Anchors. To overcome these pitfalls, we identify unit test assertions as the optimal semantic units for instantiation. Assertions are not random data points; they are crystallized checkpoints of developer intent, strategically placed to verify critical state transitions after complex operations. By targeting assertions, we filter out implementation noise and focus solely on the logical invariants of the system. We employ a Mask-and-Reason strategy: we take a valid assertion (e.g., `assert len(cache) == 5`) and mask the ground truth value (e.g., `assert len(cache) == <mask>`). This transforms the task into Abductive Reasoning: the model cannot find the answer by reading the immediate text. Instead, it must mentally reconstruct the preceding execution history, tracing data flows across files and simulating state mutations, to deduce the only logical value that satisfies the verification condition.

3.2 Foundation: Real-world Repository Curation

To ensure that our semantic anchors are grounded in reality, we curated a set of mature, actively maintained Python repositories. This stage ensures the fidelity of our benchmark, providing the complex, cross-file dependency structures that synthetic datasets lack.

3.2.1 Repository Selection

We selected mature, actively maintained, and predominantly pure Python open-source projects from PyPI and GitHub. Our selection specifically targets libraries where the core logic is implemented in Python, enabling complete line-level tracing via `sys.settrace`. Libraries with substantial C/C++/Cython extensions (*e.g.*, NumPy, Pandas, PyTorch, scikit-learn) were explicitly excluded, as their core computational routines remain opaque to Python-level instrumentation.

The selected repositories cover diverse reasoning domains and exhibit diverse complexity, with

codebases ranging from approximately 1,200 to over 775,000 lines of code (LoC). This deliberate variation allows for a comprehensive evaluation of LLMs' code reasoning capabilities across different scales and perspectives. Detailed description and statistics are shown in Table 5, Section A.

3.2.2 File-Level Semantic Filtering

Test selection prioritizes cases requiring multi-step semantic reasoning over superficial pattern matching. We manually review source code at the file level to screen all test modules. This initial stage aims to filter out low-value assertions and ensure Deterministic Verifiability. We systematically excluded test files falling into three specific categories: (1) Trivial API Validation, comprising simple attribute access (*e.g.*, `assert url.scheme == 'http'`), direct dictionary operations without cross-module dependencies (*e.g.*, `assert d.get('key') == 'value'`), and metadata introspection lacking computational reasoning (*e.g.*, `assert num_required_args(lambda x: None) == 1`); (2) Shallow Computation Patterns, including direct formula application or lookup tables (*e.g.*, `assert quote('a b') == 'a%20b'`), simple string manipulation without state complexity, and local transformations requiring no cross-module reasoning (*e.g.*, `assert x + 0 == x`); and (3) Non-deterministic or External Dependencies, such as tests relying on randomness (*e.g.*, `assert random.choice(seq) in seq`) or external timing or system state (*e.g.*, `assert time.time() > start`).

3.3 Countering Memorization: The Execution-Driven Mutation Engine

The second line of defense addresses Rote Memorization. Since powerful LLMs have likely memorized public codebases, using raw GitHub code leads to data contamination. Our Execution-Driven Mutation framework serves as an Anti-Contamination Engine, creating parallel reasoning realities that preserve logic but alter facts.

Unified Semantic and Logic Mutation. We employ a unified mutation strategy utilizing a teacher LLM. In a single step, the model performs both visual changes (*e.g.*, renaming variables, restructuring code, and removing comments) and logic changes (*e.g.*, modifying constants, input data, and parameters) (Jia and Harman, 2011). This combined approach ensures the mutated code is correct and readable, while effectively removing visual clues and invalidating memorized answers. Crucially, it strictly preserves the original API call sequence to the target library to maintain invariant reasoning complexity.

Execution-Driven Ground Truth Regeneration. To ensure the correctness of mutated tests and avoid LLMs hallucinations, we designed an Execution-Driven Assertion Reconstruction framework. First, we use Probe Injection, where the LLMs converts original assertions into print statements formatted as `DEBUG_RESULT`, turning the test into code that outputs its internal state. Next, we run this code in a controlled environment to capture the actual Runtime Values produced by the logic changes. These values serve as the ground truth, derived directly from code execution rather than model guessing. Finally, we perform Style-Consistent Assertion Generation. Using the captured values and the original code's style as guides, the LLMs reconstructs the assertions. It mimics the original writing patterns, *for example*, keeping `is` for object identity checks instead of changing to `==`, or using specific constructors like `FiniteSet(1, 2)` instead of native sets like `{1, 2}`. This ensures the new ground truth is mathematically accurate and matches the original codebase's style.

Strict Validation Gate. We enforce a strict validation process. A mutated test is considered valid only if it meets three criteria: (1) Executability, where the code runs without errors; (2) Assertion Validity, ensuring new assertions pass against the mutated logic; and (3) API Preservation, where the API call sequence matches the original test. Any variant failing these checks is discarded, ensuring all tasks in RepoReason are solvable, deterministic, and maintain original reasoning depth.

3.4 Countering Superficiality: Trace-Based Structural Filtering

The third defense targets Logical Superficiality. Even in complex repositories, many tests remain shallow. To ensure RepoReason evaluates genuine Repository-Level capabilities, we employ Trace-Based Structural Filtering (Phase 2) to identify tasks with sufficient logical depth.

Trace Collection. With the mutated test cases ready, we capture their runtime behavior using a lightweight tracing system based on Python's built-in `sys.settrace()` function, which enables line-level execution tracking. Our system records the complete sequence of function calls during the execution of the mutated tests, capturing granular

details including function names, module paths, line numbers, call order, call stack depth, function source code, input arguments, return values, and executed lines within each function. Critically, to eliminate noise, we automatically identify the test function as the entry point and focus strictly on the target library's execution path. Furthermore, to prevent infinite recursion and ensure data manageability, we enforce a maximum call depth of 3, where the test function itself is defined as depth 0.

Function-Level Structural Filtering and Complexity Grading. Instead of traditional hard thresholds, we implemented a weighted scoring system based on captured traces to identify high-density test cases. We calculated a complexity score for each test function:

$$\begin{aligned}\text{Score} = &\left(\frac{N_{\text{files}}}{4} \times 0.1\right) + \left(\frac{N_{\text{funcs}}}{15} \times 0.2\right) \\ &+ \left(\frac{N_{\text{calls}}}{30} \times 0.5\right) + \left(\frac{D}{4} \times 0.2\right)\end{aligned} \tag{1}$$

This formula prioritizes computational intensity (Call Count, 50%) and execution depth (Stack Depth, 20%), while balancing coverage breadth (Function Count, 20%) and cross-module dependencies (File Count, 10%). The baselines ($N_{\text{files}} = 4, N_{\text{funcs}} = 15, N_{\text{calls}} = 30, D = 4$) are calibrated to the empirical distribution of the target repositories. We primarily select tests rated 3 stars or higher (Score ≥ 0.30), filtering out approximately 13% of trivial test cases.

3.5 Countering Ambiguity: The Deterministic Value Protocol

To ensure reproducibility and eliminate evaluation noise, we introduce the Deterministic Value Protocol. This protocol enforces that ground-truth answers are unique and stably representable through two programmatic filtering funnels (covering ∼90% of samples):

Semantic Determinism Funnel: Automatically filters samples with inherent uncertainty, including: (1) representational drift (*e.g.*, strings containing random memory addresses); (2) external perturbations (*e.g.*, non-deterministic signals like timestamps or UUIDs); and (3) multi-solution semantics (*e.g.*, range comparisons like `assert x > 10`).

Morphological Determinism Funnel: Mandates that masked values represent concrete runtime states rather than arbitrary code text. A whitelist priority is established: Literals > Global Constants > Parameter-Resolvable Constructors. As a special case, *sympy* allows complex mathematical expressions (*e.g.*, `Rational(33, 2) - Rational(11, 2)*sqrt(3)`) as valid answers, as the expression itself serves as the standard storage form. Pure variable references or dynamic function calls are strictly rejected.

For descriptive variables used by developers (*e.g.*, `assert result == expected`), the system activates an LLM Variable Resolver to unfold abstract references into runtime-verified literal forms, ensuring evaluation is anchored to deterministic system states.

3.6 Cognitive Diagnostic Framework

To quantify the intrinsic reasoning complexity of a task, we perform dynamic program slicing (Agrawal and Horgan, 1990) rooted at the assertion to obtain the minimal causal computational subgraph along the actual execution path. Dynamic cross-function backward slicing is obtained by tracing back from the assertion along data and control dependencies within a recorded execution trace to isolate the minimal set of statements that causally influence the observed result. Based on this subgraph, we characterize reasoning intensity using three mutually orthogonal metrics rooted in cognitive load theory (Sweller, 1988).

1. Reading Load (ESV — Effective Sliced Volume): Measures the causally relevant source code volume required to process. We quantify the source volume of computational units (functions/methods) in the slice rather than simple slice line counts, as reasoning requires restoring semantic context (parameters, constraints, preconditions) within stable boundaries. Failures at high ESV signify *Comprehension Failure* rooted in *Context Overload*, where the agent loses semantic focus across the extended causal chain.

2. Simulation Depth (MCL — Mutation Chain Length): Measures the number of dynamic state update steps required to reach the target state. We treat each causally relevant statement in the dynamic slice as a state advancement atomic step and sum them weighted by execution frequency. This metric reflects the difficulty of maintaining a virtual machine state. Significant performance deterioration at high MCL indicates *Simulation Failure* stemming from a *State Tracking Deficit*, where the agent struggles to accurately synchronize variable states through complex sequences, leading to the breakdown of deep simulation.

3. Integration Width (DFI — Dependency

Fan-in): Measures the number of independent upstream logical inputs that collectively determine the critical value at an assertion. It quantifies logical inputs consumed by the execution subgraph but generated outside of it, while explicitly filtering out structural noise such as language keywords or module imports. Performance degradation at high DFI reveals an *Integration Failure* caused by an *Aggregation Deficit* in synthesizing disparate logical sources into a coherent conclusion.

4 Experimental Evaluation

To validate the diagnostic effectiveness of RepoReason, we conducted a comprehensive evaluation using state-of-the-art LLMs.

4.1 Experimental Setup

Agent Framework. We utilized OpenHands (version 0.60.0) (Team, 2025a) as our evaluation platform. Specifically, we configured the ReadOnlyAgent for all tasks. This agent configuration is restricted to file system exploration and code reading operations, preventing any modification to the repository. This setup perfectly aligns with the verification nature of RepoReason, isolating the model's ability to "read, navigate, and reason" from its ability to "edit or plan."

Model Selection. We evaluated a diverse set of frontier models representing the latest advancements in code reasoning: the **Claude Family** (claude-sonnet-4.5 (Anthropic, 2025)), the **GPT Family** (gpt-5.2 (OpenAI, 2025)), the **DeepSeek Family** (deepseek-3.1-terminus (DeepSeek-AI, 2025)), the **Kimi Family** (Kimi-K2 (AI, 2025)), and the **Qwen Family** (qwen3-coder-480b-a35b (Team, 2025b)).

Inference Configuration. To ensure reproducibility and eliminate generation randomness, we set the temperature to 0 for all model evaluations. Additionally, to accommodate the potentially lengthy reasoning chains required for repository-level tasks, we set the maximum generation length (max_tokens) to 8192 for each inference step.

4.2 Overall Performance: Accuracy across Repositories and Difficulty Levels

We report the Pass@1 accuracy of the four model families across the seven selected repositories. To control for task variance, results are stratified into Easy ($N = 1009$), Medium ($N = 838$), and Hard ($N = 645$) groups. Task difficulty is annotated per instance by an independent LLM (GPT-5.2) during the generation phase using a structured prompt based on AI reasoning complexity, independent of the evaluated models: **Easy**—basic Python knowledge, direct code reading, minimal state tracking; **Medium**—non-trivial API behavior, simple state mutations, control flow loops; **Hard**—deep domain knowledge, multi-layer dependencies, extensive state tracking, or metaprogramming. The complete classification prompt is provided in Appendix I. The total sample size is Overall ($N = 2492$).

Table 2: Overall Agent Accuracy by Difficulty (%). **Bold** = best; underline = second.

Model	Overall	Easy	Medium	Hard
Claude-Sonnet-4.5	**66.98**	**80.27**	**66.79**	**47.39**
DeepSeek-v3.1-Terminus	60.96	74.03	60.02	41.71
GPT-5.2	56.86	75.92	53.94	30.85
Kimi-K2	54.74	69.38	51.79	35.66
Qwen3-Coder-480B	50.56	61.35	49.16	35.50

Table 2 summarizes the overall performance of the agents. All models exhibit a clear performance degradation from Easy to Hard tasks, with Claude-Sonnet-4.5 maintaining the highest accuracy across all difficulty levels.

Through a detailed analysis of Table 3, we observe three distinct categories:

1. Explicit Algorithmic Patterns (Cachetools and Toolz): In repositories such as `cachetools` and `toolz`, models demonstrate robust and consistent performance, with mean accuracies ranging from 78.00% to 82.00%. These libraries primarily rely on standard algorithmic paradigms—such as cache eviction and functional data flows—characterized by high logical locality and linear execution transitions. The results indicate that LLMs have effectively mastered these procedural patterns, which currently define the cognitive comfort zone for agentic reasoning.

2. Implicit Metaprogramming Logic (Attrs and Jinja2): `attrs` and `jinja2` exhibit a significant performance stratification. Taking `attrs` as an example, which relies heavily on decorators for implicit class generation (*e.g.*, the dynamic injection of `__init__` methods), the performance gap between Claude (72.41%) and Kimi-K2 (43.10%) reveals a fundamental divergence in the internalization of the Python dynamic object model. Frontier models demonstrate the capacity to model runtime behaviors that remain invisible as explicit code

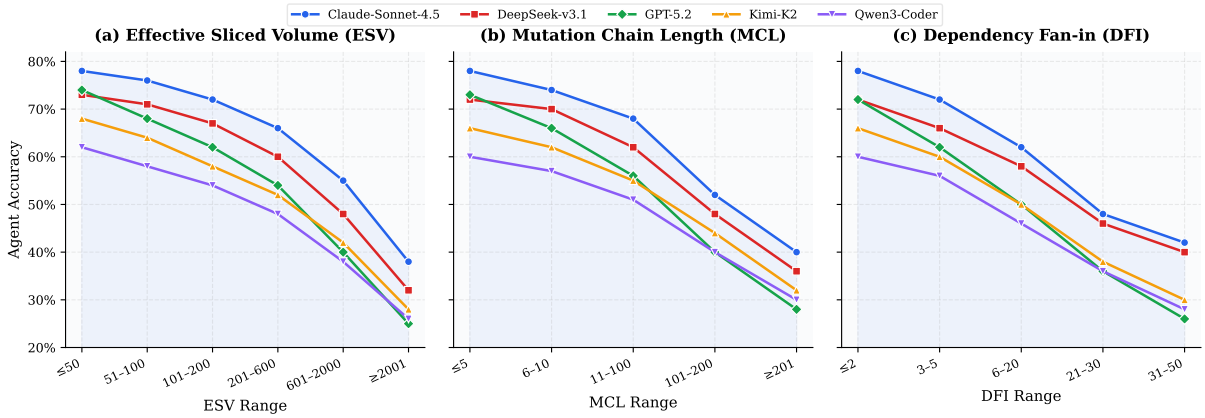

Figure 2: Performance trajectories of frontier LLM agents across three orthogonal cognitive metrics: (a) Effective Sliced Volume (ESV), (b) Mutation Chain Length (MCL), and (c) Dependency Fan-in (DFI).

lines in static text, whereas other models show a marked deficiency when logic is generated programmatically rather than defined literally.

3. High-Entropy Symbolic and Structural Reasoning (SymPy and NetworkX): `sympy` and `networkx` represent the primary reasoning ceiling for current agents. Symbolic transformations in `sympy` involve abstract, non-linear logical transitions that frequently cause reasoning chains to deviate. At the same time, `networkx` requires high structured working memory to track complex graph topologies. Claude's superior performance in `networkx` (79.77%) confirms its advanced ability to maintain accurate state snapshots during complex traversals without experiencing memory decay or structural confusion.

Table 3: Agent Accuracy Breakdown by Repository (%). **Bold** = best; underline = second.

Model	cache (193)	toolz (320)	yarl (192)	netx (404)	jinja (325)	attrs (58)	symp (1000)
Claude-4.5	**79.79**	**82.81**	78.65	**79.77**	67.46	**72.41**	**51.10**
Deep-3.1	78.24	77.19	70.83	63.37	**74.15**	62.07	45.20
GPT-5.2	78.76	72.19	**81.77**	65.10	66.46	65.52	36.00
Kimi-K2	78.76	62.81	52.08	70.54	60.92	43.10	39.60
Qwen3-Coder	70.98	62.81	59.90	57.43	55.38	43.10	37.00

4.3 Diagnostic Analysis via Cognitive Metrics

To provide a systematic white-box diagnosis of agentic code reasoning, we analyze model performance across three orthogonal cognitive dimensions: Reading Load (ESV), Simulation Depth (MCL), and Integration Width (DFI).

Figure 2 visualizes the performance trajectories of all model families as a function of the three cognitive intensity metrics. We observe a consistent and pronounced inverse relationship across all axes: accuracy systematically degrades as ESV, MCL, and DFI increase. This trend empirically validates our diagnostic framework, demonstrating that RepoReason effectively quantifies the escalation of cognitive load in repository-level reasoning and serves as a robust discriminator for architectural intelligence.

A comparative analysis of the decline slopes identifies DFI as the primary cognitive bottleneck for next-generation software agents. The slope of performance degradation is steepest along the DFI axis across all models. When DFI exceeds 20 independent upstream sources, accuracy for most models, with the exception of Claude, falls below the 40% threshold, revealing an Aggregation Deficit: agents struggle to parallelly hold and synthesize constraints from multiple disparate logical sources.

Regarding context processing, we identify a clear "Cliff Effect" in ESV. Performance remains relatively stable in lower context ranges but exhibits a sharp decline once the volume of causally relevant computational units exceeds 600 LoC. This threshold marks the transition into Context Overload, where agents fail to maintain semantic focus across the sliced causal chain. Similarly, for MCL, we observe a significant deterioration in logical consistency beyond 100 execution steps.

Table 4 provides the precise Pearson correlation coefficients for these trends. The statistical evidence confirms that while reading load and update depth are significant factors, DFI exhibits the strongest negative correlation with accuracy, reaching -0.234 for GPT-5.2. This confirms that Integration Failure is the most severe obstacle in repository-level reasoning, necessitating that future

research prioritize enhancing agents' capabilities in high-breadth information synthesis.

Table 4: Pearson Correlation between Cognitive Metrics and Accuracy. **Bold** = strongest per row.

Model	ESV	MCL	DFI
GPT-5.2	−0.188	−0.158	**−0.234**
Claude-Sonnet-4.5	−0.161	−0.122	**−0.225**
DeepSeek-v3.1-Terminus	−0.152	−0.122	**−0.157**
Kimi-K2	−0.130	−0.117	**−0.195**
Qwen3-Coder-480B	−0.148	−0.154	**−0.196**

Analysis of Table 4 reveals distinct cognitive profiles for each model family, highlighting unique architectural trade-offs. *DeepSeek-v3.1* emerges as the most architecturally balanced model, showing exceptional robustness against high Integration Width (DFI, -0.157), whereas *Claude-4.5* excels in simulation depth but remains fragile in multi-source integration (-0.225). Furthermore, while *Qwen3-Coder* demonstrates superior resilience to context volume (ESV, -0.148) and performs optimally in handling large-scale contexts, its state-tracking consistency lags behind the leaders, and *GPT-5.2* remains globally the most vulnerable model across all dimensions.

While the three metrics exhibit conceptual orthogonality, they correlate numerically in real-world codebases. To isolate the primary bottleneck, we conducted a partial correlation analysis across all models. Even after controlling for ESV and MCL, DFI maintains a highly significant partial correlation with accuracy ($r = -0.127, p < 10^{-9}$), whereas the partial correlations for ESV and MCL become statistically insignificant. This confirms DFI as the true primary barrier to repository-level reasoning.

To concretize this finding, we examine two representative high-DFI failures in detail (full cases in Appendix G). In Jinja2's `test_groupby_default` (DFI=16), the agent correctly inferred the `default` parameter semantics but failed to integrate the implicit sorting step of `groupby`, producing two disjoint groups instead of one merged group. In NetworkX's `test_all_simple_paths` (DFI=10), the agent accurately constructed the graph topology and enumerated paths but completely ignored the `cutoff=2` constraint, returning paths with 3–4 edges. Both cases exhibit a common *partial integration followed by breakage* pattern: agents successfully process initial information layers but systematically drop final constraints as the number of independent sources grows. This mechanistic evidence corroborates the statistical finding that DFI is the primary cognitive bottleneck.

4.4 Ablation Study

To quantify the impact of our benchmark design choices, we conduct two ablation studies on the DeepSeek-v3.1-Terminus agent. First, we evaluate the Execution-Driven Mutation Engine (EDME). Without EDME (evaluating on unmutated, original repository code), the agent achieves 73.15% accuracy, which drops significantly to 60.96% with EDME enabled. This exposes a Pairwise Memorization Rate ($\mathrm{MR} = \Pr(\mathrm{pass}_{\mathrm{orig}} \wedge \neg\mathrm{pass}_{\mathrm{mut}})$) of 10.71%, validating EDME's effectiveness in mitigating data contamination. Second, we assess the Structural Filtering phase. By comparing unfiltered assertions against our filtered subset, we observe substantial increases in average task complexity across all metrics: mean ESV rises from 327.6 to 393.6 (+20.1%), mean MCL from 85.3 to 93.9 (+10.1%), and mean DFI from 6.0 to 8.3 (+38.3%). This confirms that our filtering mechanism successfully isolates the most cognitively demanding reasoning tasks.

5 Conclusion

We presented REPOREASON, a white-box diagnostic benchmark for evaluating repository-level code reasoning in LLM agents. By centering on Abductive Assertion Verification and leveraging an Execution-Driven Mutation framework that treats the runtime environment as a Semantic Oracle, we ensure that tasks are both logically authentic and resistant to data contamination. Our fine-grained diagnostic system, built on dynamic program slicing, quantifies reasoning complexity through three orthogonal cognitive metrics: Reading Load (ESV), Simulation Depth (MCL), and Integration Width (DFI). Comprehensive evaluations of frontier models reveal a prevalent Aggregation Deficit, where integration width serves as the primary cognitive bottleneck, alongside a Cliff Effect in reading comprehension beyond 600 lines and significant consistency loss beyond 100 execution steps. These findings provide actionable insights for the next generation of agentic software engineering systems.

6 Limitations

Despite the rigorous design of RepoReason, we identify several areas for future refinement:

Language Scope: Currently, RepoReason focuses primarily on Python. Although Python remains the de facto standard for AI-native development and LLM research, extending this benchmark to languages such as Java, C++, and Rust is a priority for future work. We address this potential limitation by ensuring that our system architecture is built on standard profiling interfaces, which are conceptually portable to other language ecosystems, thereby enabling our white-box diagnostic paradigm to cover a broader range of programming environments.

Logical Depth Relative to Base Tests: The reasoning depth of the generated tasks is partially limited by the complexity of the original repository's test suites. We ensure foundation quality by curating mature projects with exceptionally high test coverage and complexity. The mutation engine then evolves these human-written fixed tests into vast permutations of reasoning scenarios, significantly enhancing task diversity. Incorporating adversarial test generation techniques (Shi et al., 2026) to augment the source test pool is a promising future direction.

Trace Depth Optimization: We employ a maximum call depth limit (`max_call_depth=3`) during dynamic trace extraction. Our analysis reveals that removing this limit inflates trace volume by 6.2x overall, predominantly introducing low-level infrastructure functions (*e.g.*, AST traversers, magic methods) that contribute minimally to semantic understanding. Thus, the depth limit serves as a crucial signal-to-noise ratio optimization rather than an artificial restriction on reasoning capacity.

LLM Participation in Pipeline: While our data construction pipeline utilizes LLMs, their involvement is strictly constrained to two of the five stages (mutation generation and fallback variable resolution, covering $\sim$10% of samples). Crucially, all ground-truth state values are captured through concrete code execution, not LLM generation. This hybrid approach ensures that potential model biases do not compromise the objective validity of the evaluation targets.

7 Ethics Statement

RepoReason is constructed entirely from public repositories with open-source licenses that permit research and software usage. During data collection and evaluation, we do not collect metadata regarding repository pull request authors or individual contributors, and our pipeline utilizes only publicly available information accessible via standard APIs. Our work did not involve human subject participation; we did not crowdsource or recruit human workers for any stage of the benchmark construction. All manual validation of task instances, environment setup, and metric calibration were conducted solely by the authors.

Acknowledgments

This work was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. SRFS2425-4S03 of the Senior Research Fellow Scheme and No. CUHK 14209124 of the General Research Fund).

References

Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256.

Moonshot AI. 2025. Kimi-k2: Open agentic intelligence. https://www.moonshot.cn/Kimi-K2/.

Anthropic. 2025. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Junkai Chen, Zhenhao Li, Zhiyuan Pan, Ge Li, Xing Hu, and Xin Xia. 2024. Reasoning runtime behavior of a program with LLM: How far are we? *arXiv preprint arXiv:2403.16437*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

DeepSeek-AI. 2025. Deepseek-v3.1 technical report. https://api-docs.deepseek.com/news/news250922. Technical Report for DeepSeek-3.1-Terminus.

Chunyuan Deng, Yilun Zhao, Yuzhao Heng, Yitong Li, Jiannan Cao, Xiangru Tang, and Arman Cohan. 2024. Unveiling the spectrum of data contamination in language models: A survey from detection to remediation. *Preprint*, arXiv:2406.14644.

NetworkX Developers. 2025a. Networkx: Network analysis in python. https://github.com/networkx/networkx.

PyToolz Developers. 2025b. toolz: A functional standard library for python. https://github.com/pytoolz/toolz.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *Preprint*, arXiv:2308.10620.

Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678.

Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic software issue resolution with large language models: A survey. *Preprint*, arXiv:2512.22256.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

John R Josephson and Susan G Josephson. 1994. *Abductive inference: Computation, philosophy, technology*. Cambridge University Press.

Thomas Kemmer. 2025. cachetools: Extensible memoizing collections and decorators. https://github.com/tkem/cachetools.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems.

OpenAI. 2025. Gpt-5.2 technical report. https://openai.com/index/introducing-gpt-5-2/.

Pallets. 2025. Jinja: A very fast and expressive template engine. https://github.com/pallets/jinja.

Hynek Schlawack. 2025. attrs: Python classes without boilerplate. https://github.com/python-attrs/attrs.

Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. 2026. Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. *arXiv preprint arXiv:2602.20213*.

Andrew Svetlov. 2025. yarl: Yet another url library. https://github.com/aio-libs/yarl.

John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285.

OpenHands Team. 2025a. Openhands: Code less, do more. https://github.com/OpenHands/OpenHands.

Qwen Team. 2025b. Qwen3-coder: Harnessing 480b moe for expert-level software engineering. https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct-FP8.

SymPy Development Team. 2025c. Sympy: Symbolic computing in python. https://github.com/sympy/sympy.

Danning Xie, Xuwei Liu, Chengpeng Wang, Mingwei Zheng, Jiannan Wang, Lin Tan, and Xiangyu Zhang. 2025. CORE: Benchmarking LLMs' code reasoning capabilities through static analysis tasks. *arXiv preprint arXiv:2507.05269*.

A Repository Selection and Statistical Overview

Table 5 presents the detailed statistics and core reasoning scenarios for our selected repositories, where lines of code (LoC) are measured using the `cloc` tool and specifically represent source code by excluding configuration files, documentation, and metadata.

B Other Statistics from Experiment and Dataset

B.1 Diagnostic Insights: Reasoning Scope

Figure 3 illustrates the impact of reasoning scope on agent performance. We observe a consistent performance decay as the task transitions from Intra-file Isolation to Repository-Wide Synthesis. This downward trend quantifies the Retrieval Failure described in our diagnostic framework, highlighting a significant Spatial Deficit in current agents' ability to locate and navigate long-distance dependencies across the repository architecture. Notably, cross-file reasoning tasks constitute the vast majority of RepoReason, further emphasizing its authentic repository-level nature.

Table 5: Statistics and Key Reasoning Scenarios of Selected Repositories

Repository	LoC	Tests	Core Reasoning Complexity
cachetools (Kemmer, 2025)	1.2k	88	Cache eviction (LRU/LFU/TTL), state management
yarl (Svetlov, 2025)	2.3k	369	URL parsing, normalization, codec consistency
toolz (Developers, 2025b)	3.7k	92	Func. flow, lazy evaluation, high-order logic
attrs (Schlawack, 2025)	6.3k	209	Metaprogramming, class gen, attribute validation
jinja2 (Pallets, 2025)	14k	527	Template compilation, lexical parsing, context flow
networkx (Developers, 2025a)	67k+	534	Graph traversal, centrality, complex connectivity
sympy (Team, 2025c)	776k	913	Symbolic math, solver logic, recursive calculus

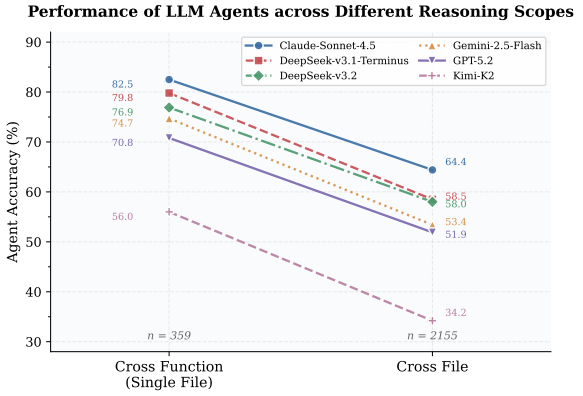

Figure 3: Performance of LLM agents across different reasoning scopes.

B.2 Dataset Characteristics: Cognitive Metrics Distribution

Figure 4 illustrates the empirical distribution of the three cognitive metrics across the RepoReason dataset. All metrics exhibit a characteristic long-tail distribution, indicating a wide range of task difficulties. The Effective Sliced Volume (ESV) distribution ($\mu = 393.6, \sigma = 706.3$) shows that a significant portion of tasks require navigating hundreds or thousands of lines of code. The Mutation Chain Length (MCL) distribution ($\mu = 93.9, \sigma = 198.0$) highlights the depth of state tracking required, while the Dependency Fan-in (DFI) distribution ($\mu = 8.3, \sigma = 7.5$) captures the integration complexity of multi-source information. This diverse distribution ensures that RepoReason provides sufficient data points across various complexity levels for fine-grained diagnostic analysis.

B.3 Partial Correlation Analysis of Cognitive Metrics

While ESV, MCL, and DFI exhibit conceptual orthogonality (measuring reading comprehension, sequential simulation, and multi-source synthesis, respectively), they naturally correlate in real-world codebases where complex tasks demand simultaneous scaling across all cognitive dimensions. To isolate the independent contribution of each metric, we conducted a partial correlation analysis on GPT-5.2 accuracy ($N = 2492$), controlling for the other two metrics in each case (Table 6).

Table 6: Partial Correlation of Each Metric with Accuracy, Controlling for the Other Two ($N = 2492$, GPT-5.2)

Metric	Controlled For	Partial r	p-value
ESV	MCL, DFI	-0.004	0.835
MCL	ESV, DFI	-0.021	0.310
DFI	ESV, MCL	$\mathbf{-0.127}$	$\mathbf{1.03 \times 10^{-9}}$

After controlling for ESV and MCL, DFI maintains an independent and highly significant negative correlation with accuracy ($r = -0.127, p < 10^{-9}$), whereas the partial correlations for ESV ($r = -0.004$) and MCL ($r = -0.021$) become statistically insignificant. This confirms that integration width constitutes a unique cognitive bottleneck distinct from general task difficulty or reading load.

B.4 Robustness Analysis: Temperature Sensitivity

To verify the stability of our findings against generation hyperparameters, we conducted a supplementary experiment using Claude-Sonnet-4.5 under both greedy decoding (`temperature=0`) and default sampling (`temperature=1.0`). Table 7 presents the repository-level accuracy changes. The overall accuracy shift is minimal (+1.00pp). Crucially, the Pearson correlations between accuracy and the three cognitive metrics remain remarkably stable (Table 8), demonstrating that the observed aggregation deficit and cliff effects are structural failures of the model architecture, rather than artifacts of decoding strategies.

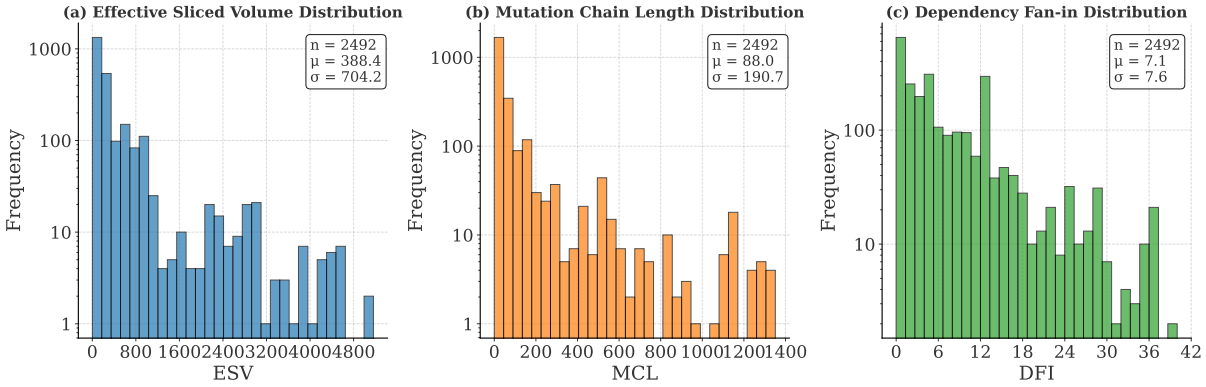

Figure 4: Empirical distribution of the three cognitive metrics across the RepoReason dataset: (a) ESV, (b) MCL, and (c) DFI.

Table 7: Claude-Sonnet-4.5 Accuracy (%): `temperature=0` vs. `1.0`

Repository	$T = 0$	$T = 1.0$	Δ **(pp)**
`attrs`	72.41	74.14	+1.73
`cachetools`	79.79	79.79	0.00
`jinja2`	67.46	68.92	+1.46
`networkx`	79.77	80.20	+0.43
`sympy`	51.10	52.70	+1.60
`toolz`	82.81	83.44	+0.63
`yarl`	78.65	80.73	+2.08
Overall	**66.98**	**67.98**	+1.00

Table 8: Stability of Metric-Accuracy Pearson r

Correlation	$T = 0$	$T = 1.0$	Δ
ESV – Accuracy	−0.158	−0.170	−0.012
MCL – Accuracy	−0.137	−0.142	−0.005
DFI – Accuracy	−0.218	−0.229	−0.011

C Technical Specification of the Agentic Search Framework

This section details the implementation of the Agentic Search evaluation framework, focusing on the specialized prompt engineering for the `ReadOnlyAgent` and the robust evaluation mechanisms used to verify repository-level reasoning.

C.1 Agentic Search Prompting Strategy

The framework utilizes a template-driven approach to construct tasks for the Agent. As illustrated in Box 1, the prompt is structured to provide necessary execution context while enforcing strict output constraints.

Agentic Search Task Prompt Template

Task: Fill in the blank in the Python test code.

Blank Assertion:
`[Blanked_Assertion_Statement]`

Hint:
[Type_Specific_Hint] (e.g., "The answer MUST be a **string literal**" or "MUST be a **constructor call** from the source code").

Instructions:
The '___' in the assertion represents a missing expression (variable name, function call, or literal).

You are a `ReadOnlyAgent`. You can read files and search the code in the repository to infer what this expression should be.
Note: Test function code is not visible in the repository, but you can explore the source code functions called by the test and reason to complete the fill-in-the-blank task.

Context:
- **Test function source code:**

```
[Source_Code_with_Question_Marker]
```

- **Input parameters:** `[JSON_Serialized_Inputs]`
Note: The missing expression is likely related to the input parameters or variables defined in the test or the code being tested.

Goal:
Find the exact expression that should replace '___'.
When you are confident, use the `finish` tool with the answer.

IMPORTANT: The finish tool's content MUST contain **ONLY** the answer expression itself, nothing else. Examples:

- If the answer is a variable: finish with content `"result"`
- If the answer is a function call: finish with content `"len(items)"`
- If the answer is a string: finish with content `'"hello"'`
- If the answer is an attribute access: finish with content `"C.__attrs__attrs__[1].validator"`

Do NOT include explanations, reasoning, or any other

text in the finish tool content. Only the answer expression.

C.1.1 Concrete Example: Literal Integer Answer

We present a real prompt instance from the cachetools repository to illustrate the complete Agentic Search prompt structure. This example demonstrates a literal integer answer (not a variable reference), where the Agent must infer the value 2 by reasoning about cache operations.

Real Agentic Search Prompt Example

Task: Fill in the blank in the Python test code.

Blank Assertion:
`assert ___ == len(cache)`

Hint:
CRITICAL REQUIREMENT: The answer MUST be a literal integer value.

IMPORTANT: Return ONLY the literal value, NOT an expression or variable name.

Type-specific examples:
- Boolean: True or False (NOT 'x == y' or 'bool(x)')
- String: "hello" (NOT 'str_var' or 'x.name')
- Number: 42 (NOT 'num_var' or 'len(x)')
- List: [1, 2, 3] (NOT 'list_var' or 'list(x)')
- Dict: {"a": 1} (NOT 'dict_var' or 'dict(x)')

Instructions:
The '___' in the assertion represents a missing expression (variable name, function call, or literal).

You are a `ReadOnlyAgent`. You can read files and search the code in the repository to infer what this expression should be.
Note: Test function code is not visible in the repository, but you can explore the source code functions called by the test and reason to complete the fill-in-the-blank task.

Context:
- **Test function source code:**

```
def test_update(self):
    cache = self.Cache(maxsize=2)

    cache.update({1: 1, 2: 2})
    assert ___ == len(cache)  # Question
    assert ___ == cache[1]
    assert ___ == cache[2]

    cache.update({1: 1, 2: 2})
    assert ___ == len(cache)
    assert ___ == cache[1]
    assert ___ == cache[2]

    cache.update({1: "a", 2: "b"})
    assert ___ == len(cache)
    assert ___ == cache[1]
    assert ___ == cache[2]
```

- **Input parameters:** `{"self": "<LRUCacheTest>"}`

Note: The missing expression is likely related to the input parameters or variables defined in the test or the code being tested.

Goal:
Find the exact expression that should replace '___'.
When you are confident, use the `finish` tool with the answer.

IMPORTANT: The finish tool's content MUST contain **ONLY** the answer expression itself, nothing else. Examples:
- If the answer is a variable: finish with content `"result"`
- If the answer is a function call: finish with content `"len(items)"`
- If the answer is a string: finish with content `'"hello"'`
- If the answer is an attribute access: finish with content `"C.__attrs__attrs__[1].validator"`

Do NOT include explanations, reasoning, or any other text in the finish tool content. Only the answer expression.

Analysis. In this example, the Agent must:

1. Understand that `cache.update({1: 1, 2: 2})` adds two key-value pairs to a cache with `maxsize=2`
2. Reason that after the update, `len(cache)` equals 2
3. Infer that the blank should be filled with the literal integer 2
4. Use the `finish` tool with content `"2"` (without quotes in the tool content, as it's a numeric literal)

This example demonstrates how the prompt guides the Agent to reason about runtime behavior while enforcing strict output constraints for evaluation.

C.2 Evaluation and Comparison Engine

Given the syntactic flexibility of Python, the framework employs a multi-criteria comparison engine to verify the Agent's output against ground truth.

C.2.1 Mathematical Equivalence ($\mathcal{M}$)

For tasks involving symbolic expressions, the engine utilizes the `SymPy` library to verify mathematical identity. Correctness is defined as:

$$\begin{aligned}\text{Correct} \iff &\text{Simplify}(P - G) = 0 \\ &\vee \text{Expand}(P) = \text{Expand}(G)\end{aligned} \tag{2}$$

where P is the Agent's prediction and G is the ground truth. This multi-path verification prevents false negatives caused by structural variations.

C.2.2 Structural Normalization ($\mathcal{S}$)

The engine applies recursive normalization to handle implementation-specific data structures:

- **Wrapper Stripping:** Automatically unwrap custom container types (e.g., `D({...})` $\rightarrow$ `{...}`).
- **Collection Alignment:** Standardize whitespace and element ordering in complex literals.
- **Object Mapping:** Map specialized collections like `defaultdict` to standard dictionary equivalents.

C.2.3 Heuristic Semantic Matching

To bridge the gap between precise syntax and reasoning intent, the following heuristics are applied:

- **Constructor Awareness:** Recognizes class instantiations required by the logic (e.g., `ClassName(args)`).
- **Path Qualification:** Standardizes module paths, accepting partially qualified references if unambiguous.
- **Quote Invariance:** Neutralizes differences in quote types (single, double, or triple).

C.3 Execution Management

The Agent is executed in a sandboxed environment with a strict 1800-second timeout. All tool invocations are captured in a serialized JSON sequence for auditability. An incremental checkpointing mechanism is implemented to ensure data persistence during large-scale evaluations.

D Countering Memorization via an Execution-Driven Mutation Engine

This appendix provides detailed implementation specifications to ensure full reproducibility of our Execution-Driven Mutation Engine (EDME). We present the system's I/O contracts, hyperparameters, algorithmic components, and concrete examples from real mutation instances.

D.1 Reproducibility Artifacts and I/O Contracts

Artifacts. Each mutation instance is serialized as a JSON record containing: (i) the original test excerpt, (ii) the probe-injected intermediate code, (iii) captured runtime values, (iv) the final mutated test, and (v) validation metadata (executability, assertion validity, and API preservation).

Input Contract. The system takes as input:

- A unit-test function (source text)
- Extracted API call signature list (Section D.3)
- Original assertion list with their structural patterns
- Repository-specific execution environment configuration

Output Contract. The system emits a mutated test function that satisfies three mandatory properties:

1. **Executability**: runs without errors in the target environment
2. **Assertion Validity**: all reconstructed assertions pass
3. **API Preservation**: maintains the original API call sequence

Metadata Schema. Each output record includes:

- `validation_status`: `valid` or `invalid`
- `api_calls_preserved`: boolean flag
- `execution_driven_assert`: nested object with probe results
- `mutation_attempts`: number of LLM retries
- `execution_time`: total processing time in seconds

Output Structure.

- Structured artifacts capturing per-repository mutation records (JSON)
- Aggregated statistics summarizing successful mutations
- Logs containing execution traces and debug output

Table 9: Key hyperparameters and hard constraints for EDME.

Item	Setting / Constraint
Teacher model	`gpt-5.2` (single-step unified mutation)
Temperature	0.7 (balancing creativity and correctness)
Max tokens	16384 per mutation request
Probe format	*print(f"DEBUG_RESULT: {expr}")* (mandatory f-string)
Max mutation retries	3 (error-feedback loop)
Execution timeout	180s per trial
Validation gate	All of: executability, assertion validity, API preservation
Hard Constraints (enforced via prompt):	
No signature drift	Function signature must not change (no new args/defaults)
No new imports	Must reuse existing imports/helpers in the file
Preserve parametrize	*pytest.mark.parametrize* decorators must remain intact
Style consistency	Match original assertion patterns (e.g., `is` *vs.* `==`)

D.2 Detailed Hyperparameters and Constraints

Table 9 lists the key hyperparameters and hard constraints required for faithful reproduction.

D.3 API-Call Sequence Preservation Mechanism

To prevent shortcutting and preserve the reasoning pathway, we enforce that the mutated test retains the original API call sequence to the target library.

Extraction Phase. We traverse the original test's Abstract Syntax Tree (AST) to extract a list of call signatures:

- Free function calls: `solve(...)`, `simplify(...)`
- Method calls: `obj.method(...)`, `expr.rewrite(...)`
- Constructor calls: `FiniteSet(...)`, `Rational(...)`

This extraction is performed by traversing the AST and collecting all callable expressions.

Validation Phase. During validation, we check that each recorded call signature appears in the mutated code as a callable occurrence. The system constructs a regex pattern for each call (matching the function/method name followed by an opening parenthesis) and verifies its presence. Any candidate that removes, substitutes, or reorders these calls is rejected.

Result Recording. The validation result is written to the metadata. Only mutations that preserve API calls proceed to the final output dataset.

D.4 Execution-Driven Mutation and Assertion Reconstruction

EDME employs a three-stage pipeline that first generates mutated code with embedded probes, then executes it to capture ground-truth values, and finally reconstructs assertions from the captured runtime values. This approach eliminates model hallucination by deriving truth from actual code execution.

D.4.1 Stage 1: Comprehensive Mutation with Probe Generation

In a single LLM call, the system performs both visual/logical mutations and probe injection simultaneously. The LLM is instructed to generate mutated code that directly replaces all original assertions with `DEBUG_RESULT` print statements, combining mutation and probe generation in one unified step.

Comprehensive Mutation Process. The mutation prompt requires the LLM to perform two tasks in parallel:

1. **Visual mutations**: Rename local variables, restructure control flow, change visual appearance
2. **Logical mutations**: Change literal constants, modify input parameters, perturb data flow
3. **Probe injection**: Replace each assertion of the form *assert expression == expected_value* with *print(f"DEBUG_RESULT: {expression}")*

This unified approach ensures that the mutated code is immediately executable as a state probe, with all mutations and probe placements completed in a single LLM generation step.

Format Enforcement. The probe format is strictly enforced via prompt constraints and post-generation validation:

- Must use f-string syntax (not *print("DEBUG_RESULT:", x)*)
- Must include the exact prefix `DEBUG_RESULT:`
- Multi-line expressions must be properly formatted

Automatic Correction. If the LLM generates incorrect probe format (e.g., using comma-separated print), the system automatically corrects it via regex replacement. If the generated code lacks `DEBUG_RESULT` outputs, the system triggers a retry with explicit feedback.

D.4.2 Stage 2: Execution-Driven Value Capture

After generating the mutated code with embedded probes, we execute it in a controlled environment and parse the stdout stream to extract `DEBUG_RESULT` payloads. This execution step captures the actual runtime values produced by the mutated logic, which serve as ground truth for assertion reconstruction.

Execution Environment. The execution is performed with the following setup:

- Isolated temporary file in the original test directory (to support relative imports)
- Repository-specific virtual environment (configured for the target project)
- Configured `PYTHONPATH` to include necessary modules
- 180-second timeout per execution

Output Parsing. The parser handles multiple output formats:

1. Standalone line: `DEBUG_RESULT: value`
2. Embedded in pytest output: ... *DEBUG_RESULT: value* ...
3. Multi-line structures: continues collecting lines until brackets/braces are balanced

Strict Mode. During probe execution, *check_debug_result=True* enforces that at least one DEBUG_RESULT must be found. Execution success without probe output is treated as failure, triggering a retry with error feedback.

D.4.3 Stage 3: Assertion Reconstruction from Runtime Values

Using the captured runtime values from Stage 2 as ground truth, a second LLM call regenerates assertions while matching the original test's idioms. The LLM converts each DEBUG_RESULT output back into an appropriate assertion statement, ensuring style consistency with the original test.

Style Matching Rules. The LLM is instructed to preserve:

- Comparison operators: `is` *vs.* `==` *vs.* `!=`
- Constructor preferences: `S.Half` *vs.* `0.5`, `Rational(a,b)` *vs.* `a/b`
- Collection types: `FiniteSet(...)` *vs.* `{...}`, `[...]` *vs.* `(...)`
- Assertion structure: single-line *vs.* multi-line, with/without intermediate variables

Error-Driven Refinement. The assertion reconstruction supports iterative refinement:

1. Generate assertions from captured runtime values
2. Replace DEBUG_RESULT statements with the generated assertions
3. Execute the final mutated code to validate all assertions pass
4. If validation fails, append error message to conversation history
5. Retry assertion generation with error feedback (up to 3 attempts)

D.5 Strict Validation Gate

A mutated test is accepted *only if* it satisfies all three validation criteria simultaneously.

Criterion 1: Executability. The probe-injected code must:

- Run without syntax errors or runtime exceptions
- Complete within the 180-second timeout
- Produce at least one DEBUG_RESULT output

Criterion 2: Assertion Validity. The final mutated code with reconstructed assertions must:

- Execute successfully (pytest returncode = 0)
- Pass all assertions against the mutated logic

Criterion 3: API Preservation. The mutated code must:

- Contain all original API call signatures
- Maintain the same call sequence (order may vary within independent statements)

Gate Enforcement. The validation status is recorded in metadata as `validation_status`. Only mutations with `validation_status="valid"` are included in the final dataset. The batch validation routine skips non-valid entries during final verification.

D.6 Key Prompt Templates

This subsection presents the essential prompt templates used in EDME's LLM-based mutation and assertion reconstruction stages.

D.6.1 Mutation Prompt (Stage 1)

The mutation prompt enforces hard constraints while guiding the model to perform both visual and logical mutations. The system prompt emphasizes:

System Prompt (Excerpt).

- **Format requirements**: All DEBUG_RESULT outputs must use f-string format: *print(f"DEBUG_RESULT: {value}")*
- **Visual mutations**: Rename local variables, restructure control flow, change visual appearance
- **Logical mutations**: Change literal constants, modify input parameters, perturb data flow
- **API preservation**: Maintain the exact API call sequence specified in constraints
- **Signature preservation**: Function signature must remain unchanged (no new args/defaults)
- **Import reuse**: Do not add new imports; reuse existing imports/helpers

User Prompt (Excerpt). The user prompt provides:

- Target function name and context (decorators, class definition if applicable)
- API call constraints list (must be strictly preserved)
- Original test code with all assertions
- Available imports and helper functions
- Explicit instruction to replace all assertions with DEBUG_RESULT outputs

D.6.2 Assertion Reconstruction Prompt (Stage 3)

The assertion reconstruction prompt guides the model to convert captured runtime values into style-consistent assertions.

System Prompt (Excerpt).

- **Style matching**: Strictly follow original assertion patterns (operators, constructors, constants)

Algorithm 1 Execution-Driven Mutation Engine (EDME)

Input: Original test T, API inventory $\mathcal{A}$, configuration C
Output: Mutated test T' with metadata M, or $\perp$ if no candidate passes validation

attempts $\leftarrow 0$
while *attempts* $< C$.*max_retries* **do**
  attempts $\leftarrow$ *attempts* $+ 1$
  Stage 1: Probe-aware mutation
  $T_{\text{probe}} \leftarrow$ MUTATEWITHPROBES$(T, \mathcal{A})$
  if $\neg$ HASDEBUGPROBE(T_{probe}) **then**
    continue ▷ Missing probe
  end if
  Stage 2: Runtime capture
  $(exec, V) \leftarrow$ EXECUTEPROBES$(T_{\text{probe}}, C.timeout)$
  if $\neg$ *exec* **then**
    continue ▷ Execution failed
  end if
  Stage 3: Assertion reconstruction
  $T' \leftarrow$ RECONSTRUCTASSERTIONS$(T_{\text{probe}}, V,$ STYLEOF$(T))$
  assert_ok $\leftarrow$ EXECUTEFINAL$(T', C.timeout)$
  if $\neg$ *assert_ok* **then**
    continue ▷ Assertion validation failed
  end if
  api_ok $\leftarrow$ VERIFYAPI$(T, T', \mathcal{A})$
  if $\neg$ *api_ok* **then**
    continue ▷ API preservation failed
  end if
  $M \leftarrow$ RECORDMETADATA(`valid`, *api_ok*, V)
  return (T', M)
end while
$M \leftarrow$ RECORDMETADATA(`invalid`, **false**, $\emptyset$)
return $(\perp, M)$

- **Format preservation**: Maintain `is` *vs.* `==`, `Rational(a,b)` *vs.* `a/b`, collection types
- **No markdown**: Return pure Python code only, no explanations
- **Type conversion**: Parse runtime values and convert to original style (e.g., `Rational(2, 3)` instead of `0.666...`)

User Prompt (Excerpt). The user prompt provides:

- Captured runtime values (from `DEBUG_RESULT` outputs) as JSON array
- Original assertion patterns (for style reference)
- Code with `DEBUG_RESULT` statements to convert
- Available imports context
- Error feedback (if previous attempt failed)

D.7 End-to-End Algorithm

Algorithm 1 presents the complete EDME pipeline in pseudocode.

D.8 Concrete Example from Real Data

We present a real mutation instance from our dataset to illustrate the complete EDME pipeline. This example is taken from the SymPy repository, specifically the `test_rewrite_trig` function from the solvers test suite.

This instance demonstrates:

- Fact-level perturbation (changing coefficients from $2, 1$ to $3, 2$)
- API preservation (`solve`, `sin`, `cos`, `atan` all retained)
- Execution-driven truth (runtime value `[atan(2/3)]` captured)
- Style consistency (using `Rational(2, 3)` instead of `0.666...`)

Mutation Analysis. The key transformations in this example:

1. **Visual mutation**: Variable x renamed to probe $= x \cdot (S.\text{One} + S.\text{Zero})$ (semantically equivalent but visually distinct)
2. **Logic mutation**: Coefficients changed from $(2, -1)$ to $(3, -2)$, invalidating the original answer $\arctan(1/2)$
3. **Probe execution**: Captured runtime output `[atan(2/3)]` from the mutated equation
4. **Style preservation**: Reconstructed as `Rational(2, 3)` (matching SymPy's preference for exact rationals) instead of `0.6666...`

Validation Results. This mutation passed all three validation criteria:

- **Executability**: Probe code ran successfully, producing *DEBUG_RESULT: [atan(2/3)]*
- **Assertion validity**: Final code passed pytest with the reconstructed assertion
- **API preservation**: All four API calls (`solve`, `sin`, `cos`, `atan`) present in original order

E Countering Ambiguity via the Deterministic Value Protocol

This appendix describes the Deterministic Value Protocol (DVP). DVP enforces hard constraints on ground-truth answers during task generation. The goal is to remove evaluation noise caused by (i) unstable string representations, (ii) environment-dependent nondeterminism, and (iii) multi-solution semantics under strict matching.

Figure 5: Real EDME instance from SymPy. Left: original test with memorized constants. Middle: probe-injected candidate with mutated coefficients and structured debug output. Right: final assertion reconstructed from runtime ground truth while keeping the preserved API pathway.

Original Test

```
def test_rewrite_trig():
    # Original equation
    assert solve(
        2*sin(x) - cos(x),
        x
    ) == [atan(S.Half)]
```

Probe-Injected

```
def test_rewrite_trig():
    # Mutated equation
    probe = x*(S.One + S.Zero)
    print(
        f"DEBUG_RESULT:
      {solve(3*sin(probe) -
      2*cos(probe), probe)}"
    )
```

Final Mutated

```
def test_rewrite_trig():
    # Reconstructed assertion
    probe = x*(S.One + S.Zero)
    assert solve(
        3*sin(probe) - 2*cos(probe),
        probe
    ) == [atan(Rational(2, 3))]
```

E.1 Artifacts and I/O Contracts

Input. DVP consumes one trace-derived test function and its assertions.

Output. For each accepted assertion, DVP emits one fill-in-the-blank instance with the masked assertion and the ground-truth answer.

Discard policy. If no mask position can satisfy DVP constraints, the sample is discarded.

E.2 System Overview

DVP is programmatic-first and fallback-last. It applies two strict "programmatic parsing funnels" and uses an LLM fallback only under a narrow boundary.

- **Funnel 1 (Semantic Determinism):** remove nondeterminism and multi-solution semantics.
- **Funnel 2 (Answer Morphological Determinism):** keep only answer forms that are stable and explicit.

LLM fallback is invoked only when the assertion is structurally maskable but the extracted answer is a variable reference. In this case, the model is restricted to a *variable resolver*. The resolved result must pass Funnel 1 and Funnel 2 again.

E.3 Funnel 1: Semantic Determinism

Funnel 1 rejects candidates that can change across runs or admit multiple valid solutions.

(1) Representational drift. DVP rejects unstable string forms. Typical patterns include:

- object repr addresses: `<... at 0x[0-9a-fA-F]+>`
- explicit memory addresses: `0x[0-9a-fA-F]+`

(2) External perturbation. DVP rejects contexts that rely on runtime-dependent signals (randomness, timestamps, UUIDs). The detector checks tokens such as `random`, `uuid`, `time.time`, `datetime.now`, and `date.today`. If matched, the sample is discarded with a reason like *nondeterministic_in_test:....*

(3) Multi-solution semantics. DVP rejects mask positions that admit multiple satisfying values under strict comparison. In operator masking, the system rejects >, <, >=, <=, and !=. For ==, DVP also rejects reflexive variable forms (e.g., `x == y`).

Approximate comparisons. DVP drops approximate floating comparisons that are sensitive to platform differences, including `pytest.approx`, `assertAlmostEqual`, `assert_allclose`, `allclose`, and `isclose`.

E.4 Funnel 2: Answer Morphological Determinism

Funnel 2 ensures the blank evaluates a concrete runtime state, not a name binding. The system assigns priorities to different answer forms (1 is best) or rejects them (priority 0), using AST-based type inference.

Whitelist with priorities. Accepted answer forms are:

- **Priority 1: Literals** (numeric, string, boolean constants, and container literals). Always accepted.
- **Priority 2: Global constants** (uppercase identifiers following naming conventions).
- **Priority 3: Stable attribute access** (e.g., `response.status_code`, `S.NaN`), without call parentheses.
- **Priority 4: Parameter-resolvable constructors** (`ClassName(...)`), only when all arguments are literals. Empty built-in constructors `set()`, `list()`, `dict()`, `tuple()`, `frozenset()` are treated as Priority 1.
- **Priority 5: Complex value expressions** only for symbolic libraries (e.g., SymPy). As a special case, symbolic computation libraries like SymPy allow complex mathematical expressions (e.g., *[Rational(33, 2) - Rational(11, 2)*sqrt(3)]*)

as valid answers.

Rejections. DVP rejects:

- pure variable references
- dynamic function calls (`func()` or `obj.method()`) unless it is a whitelisted constructor
- type-casts with variable arguments (e.g., `set(cache)`)
- unpack operators (`*args` / `**kwargs`)

E.5 Static Analysis vs. LLM Fallback Boundary

DVP has a strict boundary between static parsing and fallback.

- If programmatic parsing cannot find a valid mask position that passes both funnels, the sample is discarded.
- If the mask position is valid but the extracted answer is a variable name, fallback is allowed.

In fallback mode, the model acts only as a *variable resolver*: it must unfold the variable into a concrete literal/constant/constructor/attribute form derived from the execution context. The resolved result is re-validated through both funnels.

E.5.1 Variable Resolver Prompt

When fallback is triggered, the LLM receives a prompt emphasizing runtime behavior reasoning:

Key Requirements.

- **Not code completion**: Task is to infer runtime behavior based on trace information
- **Answer format**: Must be a specific constant value or variable name (not complex expressions)
- **String quoting**: String values must be wrapped in double quotes in JSON
- **Identity *vs.* equality**: Distinguish `is` (identity) from == (equality)
- **Hint quality**: Must guide reasoning without leaking the answer

Prompt Structure. The prompt includes:

- Complete assert statement with context
- Full test function code
- Execution flow (trace information with function calls)
- Input parameters
- Design requirements emphasizing runtime behavior reasoning
- Answer requirements specifying format constraints

Algorithm 2 Deterministic Value Protocol (DVP)

Input: Trace record R, source code S
Output: Deterministic fill-in-the-blank set $\mathcal{Q}$

Trace parsing
$T \leftarrow \textsc{ExtractTraceInfo}(R)$
$A \leftarrow \textsc{ParseAssertions}(S)$
$\mathcal{Q} \leftarrow \emptyset$
Candidate evaluation
for each assertion a in A **do**
  $c \leftarrow \textsc{MaskCandidate}(a, T)$
  if $c = \perp$ **then**
    continue ▷ No deterministic mask
  end if
  if $\neg\, \textsc{SemanticDeterminism}(c, a, T)$ **then**
    continue ▷ Fails Funnel 1
  end if
  if $\textsc{MorphologyPriority}(c, T) = 0$ **then**
    continue ▷ Fails Funnel 2
  end if
  if $\textsc{IsVariable}(c)$ **and** $\textsc{FallbackAllowed}$ **then**
Fallback resolution
    $c' \leftarrow \textsc{ResolveVariable}(c, T)$
    if $\neg\, \textsc{SemanticDeterminism}(c', a, T)$ **then**
      continue
    end if
    if $\textsc{MorphologyPriority}(c', T) = 0$ **then**
      continue
    end if
    $c \leftarrow c'$
  end if
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\textsc{EmitQuestion}(a, c)\}$
end for
Batch emission
return $\mathcal{Q}$

E.6 End-to-End Pseudocode

Algorithm 2 summarizes the end-to-end Deterministic Value Protocol.

E.7 Concrete Example from Real Data

Funnel 1 rejects unstable object repr strings. For example, in a representative dataset we observe inputs containing:

```
<tests.test_dunders.TestEqOrder object at
    0x7f6a9d5d38f0>
```

DVP rejects such answers, ensuring the final ground truth is stable under strict comparison.

Funnel 2 rejects non-explicit truths such as pure variable names (e.g., `result`) and dynamic calls (e.g., `func(value)`), which would otherwise permit shallow syntactic splicing without reasoning about runtime state.

F Cognitive Metrics Computation

This section discloses the inputs, assumptions, derivations, and outputs behind the Cognitive Metrics so that any researcher can reproduce ESV/MCL/DFI in an isolated environment. The main paper merely consumes the reported scores; implementation details, hyperparameters, algorithm descriptions, and samples live in this appendix.

F.1 Artifacts and Input Contracts

Trace Record. Each JSONL entry represents a single test execution with `function_calls` (array of `call_order`, `function_name`, `file_path`, `function_source_code`, `executed_lines`), a condensed `execution_flow`, and assertion metadata (`assert_line` counted from the dedented body, `line_offset` for the absolute first line).

Collector Configuration. All experiments fix `max_trace_size=1000`, `max_call_depth=3`, and `include_test_files=True`, with optional `target_modules` to bound observation. The configuration is embedded in `input_data` together with the `test_function` identifier.

Output Contract. The metric calculator consumes a single trace and emits `esv`, `mcl`, `dfi`, plus diagnostic sets such as `relevant_calls`, `relevant_lines`, and `source_variables`, enabling verification without rerunning the slice.

F.2 Interprocedural Slice and Evidence Harvesting

The implementation transforms a single trace into slice evidence via a strict pipeline so that every metric consumes the same structured input.

1. **Trace Normalization**: Build a `call_index` and `line_to_call` map over `function_calls` so each absolute line immediately resolves to its `call_order`. Empty frames are discarded and the entry frame is cached as `entry_call` to anchor later steps.
2. **Assertion Anchor Extraction**: *_extract_assert_targets_with_beniget* dedents the entry source, uses beniget def-use chains to enumerate every `gast.Name` read on the assertion line, and captures the call expressions that appear in the same line. A regex fallback is used only if the analysis fails, yielding both the target variable set and the function calls triggered by the assertion.
3. **Worklist Seeding**: Targets are enqueued as $(0, \text{targets}, \text{"assert\_vars"})$, while assertion-triggered callees are enqueued as $(\text{call\_order}, \{\text{"\_\_return\_\_"}\}, \text{"called\_from\_*"})$ so their returns are tracked through a synthetic placeholder. Decorator/dynamic heuristics are evaluated before seeding to attach `original_name` and `is_decorator` flags, preventing double counting.
4. **Frame-level Resolution**: The LIFO worklist pops $(\text{call\_order}, \text{target\_vars}, \text{context})$, retrieves `call_info`, converts relative assertion lines to absolutes (entry uses `line_offset`, other frames use their own `line_number`), and invokes *_slice_within_function* to obtain `relevant_lines`, `mutations`, `source_vars`, and `function_calls`. Execution counts are accumulated into the global `relevant_lines` map.
5. **Dependency Expansion**: Function-level `function_calls` are matched against the `call_index`; matched frames are re-enqueued with $\{\text{"\_\_return\_\_"}\}$ targets, while missing matches are captured in `untraced_function_calls` to expose trace coverage holes.

The resulting `relevant_calls`, `relevant_lines`, and `source_variables` form the unique evidence set for all metrics. Section F.3 details the fine-grained *_slice_within_function* solver, and Section F.4 derives ESV/MCL/DFI from that evidence.

F.3 Worklist Resolution Strategy

calculate_interprocedural_slice drives *_slice_within_function* with a strict LIFO worklist so that each frame undergoes AST-level backward slicing.

Per-frame pipeline.

1. **Source Canonicalization**: Fetch `function_source_code`, dedent it via `textwrap.dedent`, and build `abs_to_rel`/`rel_to_abs` maps so entry and nested frames share one coordinate system.
2. **Executed-line Alignment**: Map the recorded absolute lines to the relative set `executed_rel`, ensuring non-executed statements never enter the slice.
3. **Def-use and Control Dependencies**: Invoke *ExecutionFlowParser._build_beniget_chains* to build def-use chains and run *ControlDependencyAnalyzer* to map `line_no` $\rightarrow$ *controlling lines*, feeding explicit control edges into the backward

pass.

4. **Call Extraction**: Walk the dedented AST to capture every `gast.Call` within `executed_rel`, storing its line number, callee name, and argument variables for cross-frame propagation.
5. **Backward Expansion**: Initialize `needed_vars = target_vars` and keep popping variables. For each definition found in `variable_writes`, call `add_relevant_line` to add it into `relevant_lines_rel`, close over control dependencies, and enqueue any variables read on that line. Entry frames under *context="assert_vars"* force the assertion line plus all of its controllers to stay in the slice, whereas return contexts focus on `return` statements.
6. **Source-variable Inference**: After `relevant_lines_rel` is determined, count the variables that were read but never written within that set, subtracting builtins, keywords, imported symbols, and detected callee names. *_analyze_semantic_dependencies* inspects complex expressions (e.g., dictionaries or higher-order calls) to append their true data dependencies into `source_vars`.

Propagation semantics.

- Each state key *(call_order, frozenset(target_vars), context)* ensures a frame/target pair executes at most once even under deep recursion or decorators.
- Returned `function_calls` are matched against `call_index`; hits are re-enqueued with *{"__return__"}* targets, while misses get logged in *untraced_function_calls* to expose coverage gaps.

Once the worklist drains, `relevant_calls`, `relevant_lines`, and `source_variables` are fixed, enabling direct metric computation without further heuristics.

F.4 Metric Definitions and Fixed Assumptions

Every metric uses the same interprocedural slice. Let $\mathcal{F}_{\text{rel}}$ denote the relevant functions, $\mathcal{L}_{\text{rel}}$ the multiset of relevant lines with execution counts c, and $\mathcal{S}$ the variables read but not reassigned inside the slice.

$$\text{ESV} = \sum_{f \in \mathcal{F}_{\text{rel}}} \text{LOC}(f)$$

$$\text{MCL} = \sum_{(\ell,c) \in \mathcal{L}_{\text{rel}}} c$$

$$\text{DFI} = |\mathcal{S}|$$

- **ESV (Effective Sliced Volume)**: Deduplicate and count lines of code across $\mathcal{F}_{\text{rel}}$ to capture the unique reading load needed to justify the assertion.
- **MCL (Mutation Chain Length)**: Sum execution frequencies in $\mathcal{L}_{\text{rel}}$ to characterize the dynamic simulation depth.
- **DFI (Dependency Fan-In)**: Count elements in $\mathcal{S}$ to describe how many independent information sources feed the assertion.

F.4.1 ESV Computation Steps

1. Gather deduplicated source snippets for every function in $\mathcal{F}_{\text{rel}}$, counting executable lines only (whitespace and comments are skipped).
2. Record each function once even if it is invoked multiple times, preventing duplicate reading cost.
3. Summing all unique functions' line counts yields ESV, the minimal code volume a reviewer must inspect.

F.4.2 MCL Computation Steps

1. Maintain a map from each statement in $\mathcal{L}_{\text{rel}}$ to its execution count derived from `executed_lines`.
2. Sum the counts to obtain MCL; hot paths where the same line fires repeatedly increase the metric proportionally.
3. Identical source lines in distinct frames accumulate separately so that MCL reflects the actual dynamic chain length.

F.4.3 DFI Computation Steps

1. During dependency propagation, collect every variable that is read but never reassigned, forming the set $\mathcal{S}$.
2. Deduplicate the set so that a variable counts once regardless of how many frames reference it.
3. The cardinality of $\mathcal{S}$ equals DFI, indicating how many independent information sources must be tracked; higher values imply heavier contextual load.

The calculator exposes no tunable hyperparameters; interprocedural slicing is always enabled so that all three scores rely on identical evidence.

F.5 Representative Example

We choose the real trace `question_id=544_3` from the toolz corpus, anchored at the function `test_accumulate`. The test chains three representative scenarios—seedless accumulation, seeded accumulation, and empty-sequence fallbacks—so that every major argument pattern of `accumulate` is exercised.

Test body excerpt.

```
def test_accumulate():
    numbers = [2, 1, 4, 3, 7]
    assert list(accumulate(add, numbers)) == [2, 3, 7, 10, 17]
```

accumulate function implementation.

```
def accumulate(binop, seq, initial=no_default):
    """ Repeatedly apply binary function to a sequence,
      accumulating results

    >>> from operator import add, mul
    >>> list(accumulate(add, [1, 2, 3, 4, 5]))
    [1, 3, 6, 10, 15]
    >>> list(accumulate(mul, [1, 2, 3, 4, 5]))
    [1, 2, 6, 24, 120]

    Accumulate is similar to ``reduce`` and is good for
      making functions like
    cumulative sum:

    >>> from functools import partial, reduce
    >>> sum    = partial(reduce, add)
    >>> cumsum = partial(accumulate, add)

    Accumulate also takes an optional argument that will be
      used as the first
    value. This is similar to reduce.

    >>> list(accumulate(add, [1, 2, 3], -1))
    [-1, 0, 2, 5]
    >>> list(accumulate(add, [], 1))
    [1]

    See Also:
        itertools.accumulate :  In standard itertools for
      Python 3.2+
    """
    seq = iter(seq)
    if initial == no_default:
        try:
            result = next(seq)
        except StopIteration:
            return
    else:
        result = initial
    yield result
    for elem in seq:
        result = binop(result, elem)
        yield result
```

calculate_interprocedural_slice binds the entry frame and the `accumulate` implementation into one evidence pool: the entry frame supplies the sequences and assertions, while the library frame expands every intermediate state.

Metric derivations.

- **ESV**: Understanding how the folds evolve under different seeds requires reviewing both the test harness and `accumulate`, hence their combined 59 lines define ESV.
- **MCL**: $\mathcal{L}_{\text{rel}}$ covers the entry assertions and 11 relevant lines inside `accumulate`; the trace shows these lines are repeatedly accessed across multiple accumulation branches, with execution counts summing to 35.
- **DFI**: During propagation, the read-only dependencies expand to three symbols—`add`, `initial`, and `no_default`—each from module imports or default parameter constraints, so $|\mathcal{S}| = 3$.

Table 10: Metric values for the `test_accumulate` example.

Metric	Value	Evidence Source
ESV	59 lines	$\mathcal{F}_{\text{rel}} = \{\text{test\_accumulate}, \text{accumulate}\}$; deduplicated source from the entry test plus the library implementation totals 59 executable lines, mirroring the minimal reading load.
MCL	35	$\mathcal{L}_{\text{rel}}$ spans the entry assertions plus 11 relevant lines inside `accumulate`; the trace repeats those lines across the accumulate variants, yielding 35 total executions.
DFI	3	The slice retains three read-only dependencies—`add`, `initial`, and `no_default`—all supplied via imports or default guards, hence $\lvert\mathcal{S}\rvert = 3$.

This example shows how a single interprocedural slice lets us reason about multiple assertion branches: even when the assertions live entirely in the entry frame, the slicing framework still brings in the library implementation to keep metric computation reproducible.

G DFI Case Studies: Integration Failure in Practice

To illustrate what Integration Width (DFI) looks like in practice and how it induces Aggregation Deficits in agents, we present two concrete case studies. DFI (DFI $= |\mathcal{S}|$) measures the number of independent external variables in a dynamic slice that are read but never reassigned, reflecting how many disparate information sources an agent must simultaneously integrate to deduce the verified state.

G.1 Case 1: Jinja2 `test_groupby_default` (DFI=16)

```
def test_groupby_default(self, env):
    template_src = "".join([
        "{% for grp, bucket in
    people|groupby('city', default='SF') %}",
        "{{ grp }}: {{
    bucket|map(attribute='name')|join(' | ')
    }}\n",
        "{% endfor %}",
    ])
    tmpl = env.from_string(template_src)
    payload = [
```

```
        {"name": "alex", "city": "SF"},
        {"name": "brian", "city": "WA"},
        {"name": "cora"},                    #
    <-- no city key
        {"name": "dana", "city": "SF"},
        {"name": "erin", "city": "WA"},
    ]
    out = tmpl.render(people=payload)
    assert out == ___  # correct: "SF: alex |
    cora | dana\nWA: brian | erin\n"
```

The 16 source variables span four context layers: template compilation (`code`, `source`, etc.), runtime data (`args`, `kwargs`), filter chain (`value`), and environment configuration (`env`, `environment`, and others). The agent produced `"SF: alex | dana\nWA: brian | erin\nSF: cora\n"` (incorrect).

Failure Mechanism: The agent correctly inferred the `default='SF'` semantics (tagging `cora` as `SF`) but produced two disjoint `SF` groups. Jinja2's `groupby` filter sorts elements before grouping (`sorted(value, key=expr)`), which merges all `SF` entries. Although the agent's trajectory showed it read the documentation for `sync_do_groupby` specifying this sorting behavior, it failed to integrate this third logical layer. The aggregation chain broke after successfully integrating the first two layers.

G.2 Case 2: NetworkX `test_all_simple_paths` (DFI=10)

```
def
    test_all_simple_paths_on_non_trivial_graph():
    H = nx.path_graph(6,
    create_using=nx.DiGraph())
    extra_arcs = [(0, 6), (2, 6), (2, 4), (6,
    5), (5, 3), (5, 4)]
    H.add_edges_from(extra_arcs)

    route_iter = nx.all_simple_paths(H, 2, [3,
    4], cutoff=2)
    observed = {tuple(step_list) for step_list
    in route_iter}
    assert observed == ___  # correct: {(2,
    3), (2, 4), (2, 3, 4)}
```

The `path_graph(6)` constructs a $0 \to 1 \to \ldots \to 5$ chain, augmented by extra arcs. The `cutoff=2` parameter restricts valid paths to those with ≤ 2 edges. The 10 source variables span graph construction, path enumeration logic, and the truncation constraint (`cutoff`). The agent predicted `{(2,3), (2,4), (2,3,4), (2,4,5,3), (2,6,5,3), (2,6,5,4), (2,6,5,3,4)}` (incorrect).

Failure Mechanism: The agent correctly built the graph topology and enumerated paths but failed to integrate the `cutoff=2` constraint, returning paths with 3 and 4 edges. This exemplifies partial integration followed by breakage: as DFI increases, agents correctly process initial information layers but increasingly drop final constraints.

H Impact of Trace Depth Limit

During dynamic trace extraction, we apply a maximum call depth limit (`max_call_depth=3`) to optimize the signal-to-noise ratio. To quantify its impact, we compared traces generated with `depth=3` versus unbounded `depth=`∞ across all repositories (Table 11).

Table 11: Trace Inflation Factor (`depth=3` vs. `depth=`∞)

Repo	Samples	Depth 3	Depth ∞	Inflation	Max Dep.
`cachetools`	51	369.7	396.7	1.1x	11
`toolz`	100	40.1	50.1	1.3x	7
`yarl`	100	4.9	9.0	1.8x	10
`networkx`	100	236.3	350.8	1.5x	14
`jinja2`	100	20.4	981.4	**48.0x**	49
`attrs`	100	9.1	11.6	1.3x	8
`sympy`	100	272.1	3186.5	**11.7x**	36
Overall	**651**	**118.5**	**736.1**	**6.2x**	–

Removing the depth limit inflates function call counts by 6.2x overall. The vast majority of added calls correspond to low-level infrastructure functions (AST traversers, iterator protocols, magic methods) that do not contribute to the high-level semantic understanding of the logic. Crucially, this depth limit applies *only* to trace extraction for metric computation; during reasoning, the evaluating agent retains unconstrained access to the complete repository codebase and execution environment.

I Difficulty Classification Protocol

To control for task variance, instance difficulty is explicitly stratified based on AI reasoning complexity. This annotation is generated by an independent evaluator LLM (GPT-5.2) during the data generation phase. The LLM acts solely as an annotator, analyzing the blanked assertion and execution trace to assign a label of `easy`, `medium`, or `hard`. The prompt template is provided in Box I.

Difficulty Classification Prompt Template

You are an expert at evaluating programming question difficulty for AI systems.
Given:

- **QUESTION:** (the blanked assertion)

- **CONTEXT:** (execution-trace context for understanding only; do NOT assume the answer is visible)

Task: Classify the instance difficulty into one of {easy, medium, hard} based on AI reasoning complexity. Use the following criteria:

- **easy**: Requires only basic Python knowledge and direct code reading. Involves linear logic, shallow abstraction, and minimal state tracking.
- **medium**: Requires understanding non-trivial API behavior, simple state mutations, control flow loops, or basic algorithms.
- **hard**: Requires deep domain knowledge (e.g., symbolic math, complex graph theory), resolving deep multi-layer dependencies, extensive state tracking, or metaprogramming.

Provide your classification exactly as one of the three keywords.