Supplementary Information: GraphSentry: Contract-Checked Graph Surgery for Budgeted LLM Reasoning DAGs

Shuang Cao[1,*], Rui Li[1,*], Ruihua Liu[1], Alexandre Duprey[1], Shuchen Ge[1]

[1]Hill Research

[*]Shuang Cao and Rui Li contributed equally.

Correspondence to: rui.li@hillresearch.ai

Contents

1 Supplementary Note 1: Equal-Budget Accounting Protocol

This section provides the complete specification of our strict equal-budget accounting protocol, enabling independent replication of all reported results.

1.1 Budget Units and Counting Rules

We define the **global budget** as the total number of LLM API calls permitted across the entire test split. A "10k-call run" means exactly 10,000 LLM calls are allowed for processing all test instances. We count the following call types:

Table S1: Budget accounting rules used throughout the paper under strict equal-budget evaluation. The table defines what counts as an LLM call under the 10k-call global budget and what is tracked as secondary cost. We explicitly count executor, mutator, repair, and re-execution calls so that search overhead is included rather than hidden. Tool calls are logged with timing and outcomes to enable audit, but do not consume the LLM-call budget; API timeouts and retries do consume budget because they reflect real client-side cost. This definition is applied identically to GraphSentry and all baselines to ensure that reported gains cannot be explained by asymmetric accounting.

Call Type	**Description**	**Cost**
Executor	LLM call to execute a reasoning node	1
Mutator	LLM call to propose a DAG edit/mutation	1
Repair	LLM call to fix a failed certificate check	1
Re-execution	LLM call to re-run downstream nodes after splice	1 per node
Tool call	Invocation of external tool (SQL, unit test, API)	0 (logged)
Timeout	API call that exceeds 15s timeout	1 (counted)
Retry	Automatic retry after transient API error	1 per retry
Cache hit	Deduplicated call via shared cache	0

1.2 Caching Strategy

All methods use a **shared, deterministic cache** keyed by model, prompt, and decoding parameters. The cache is initialized empty at the start of each task run and can be shared across instances within that run; no cache state is carried across tasks or across separate runs.

We report three cache settings. First, **Global cache** enables full cache sharing across all instances (default in main results). Second, **Instance-local cache** resets the cache between test instances. Third, **No cache** counts all calls without deduplication. Cache hit rates (BBH-Logic, 10k-call) are: GraphSentry = 46.8%, LATS-MCTS = 24.9%, OPRO = 18.2%.

1.3 Timeout and Retry Policy

We enforce a strict timeout and retry policy. The **API timeout** is set to 15 seconds per call, matching production API limits. The **Retry policy** allows at most 2 retries per call for transient errors (HTTP 429, 500, 503). Each **Retry cost** counts as 1 call. Finally, **Timeout handling** dictates that timed-out calls return an empty response and cost 1 call.

1.4 Budget Composition by Method

Table S2 shows the average budget composition for each method on BBH-Logic (10k-call run).

2 Supplementary Note 2: Mechanism Variable Estimation

2.1 Definitions

We define the mechanism variables used in our theoretical model as follows. First, $\hat{\rho}$ (invalid-attempt rate) is the fraction of candidate evaluations that fail structural validity checks before producing a final answer.

$$\hat{\rho} = \frac{\#\text{ candidates failing certificate check}}{\#\text{ total candidates evaluated}}$$

Second, $\hat{d}$ (failure lateness) is the fraction of a full evaluation's cost spent before invalidity is detected.

$$\hat{d} = \frac{\text{avg cost before failure detection}}{\text{avg cost of complete evaluation}}$$

Table S2: Average budget composition on BBH-Logic under a 10k-call global budget, reported in thousands of LLM calls. The breakdown separates executor calls from mutator, repair, and re-execution calls to make overhead explicit and auditable. Tool interactions and retries are also tracked to reflect client-side reality under timeouts and rate limits. Because totals are matched at 10.0k for every method, this table clarifies how different algorithms allocate the same budget rather than assuming equal compute implicitly. The key takeaway is that GraphSentry converts budget from wasteful executor/retry usage into targeted mutation, bounded repair, and downstream-only re-execution. Specifically, GraphSentry spends 4.2k calls on execution and 1.8k on mutation, whereas LATS-MCTS spends 6.8k on execution but incurs higher retry costs. The explicit tracking of repair calls (0.8k) shows that the repair mechanism is active but bounded, preventing runaway loops. Re-execution costs (1.6k for GraphSentry) confirm that splicing triggers downstream updates, but these are less than full re-runs. Tool usage is comparable across methods, ensuring that gains are not due to tool overuse or underuse. Finally, the retry column highlights that all methods face transient failures, but GraphSentry's lower retry count suggests better handling of invalid inputs.

Method	Executor	Mutator	Repair	Re-exec	Tool	Retry	Total
GraphSentry	4.2	1.8	0.8	1.6	1.2	0.4	10.0
LATS-MCTS	6.8	0	0	1.8	1.0	0.4	10.0
OPRO	7.2	1.4	0	0.6	0.6	0.2	10.0
ToT (BFS)	7.6	0.8	0	0.4	0.8	0.4	10.0
CoT-SC	8.4	0	0	0	1.2	0.4	10.0

Third, $\hat{\rho}\hat{d}$ (wasted compute) estimates the fraction of budget wasted on structurally invalid candidates.

2.2 Estimation Protocol

We estimate $\hat{\rho}$ and $\hat{d}$ from logged traces using the following steps. First, for each candidate DAG execution, we log all node executions with timestamps and costs. Second, we mark candidates as "invalid" if any certificate check fails before final output. Third, we compute $\hat{\rho}$ as the fraction of invalid candidates. Fourth, for each invalid candidate, we compute $d_i = \text{cost\_at\_failure}/\text{cost\_if\_complete}$. Finally, we compute $\hat{d} = \text{mean}(d_i)$ over all invalid candidates.

Table S3: Per-task estimates of the mechanism variables used in the gain model, reported as mean±std over 5 seeds. $\hat{\rho}$ is the invalid-attempt rate (fraction of evaluated candidates failing structural validity), and $\hat{d}$ is failure lateness (fraction of evaluation cost spent before invalidity is detected). The product $\hat{\rho}\hat{d}$ estimates wasted compute attributable to compositional invalidity under strict accounting. ΔAcc is the measured accuracy gain of GraphSentry over the best baseline under the same budget, and Predicted Δ is the gain predicted by the fitted linear model. The key takeaway is that tasks with higher $\hat{\rho}\hat{d}$ exhibit systematically larger gains, supporting a falsifiable mechanism explanation rather than purely empirical correlation. For instance, HotpotQA-Tools has the highest $\hat{\rho}\hat{d}$ (38.4%) and the largest gain (+10.8), while WordSort has the lowest (4.2%) and smallest gain (+1.3). The high $\hat{d}$ values across most tasks (>80%) indicate that failures typically occur late in the process, making early detection valuable. The consistency between measured and predicted gains validates the linear regression model derived from these variables. This table provides the empirical basis for the decision rule presented in the discussion.

Task	$\hat{\rho}$ (%)	$\hat{d}$ (%)	$\hat{\rho}\hat{d}$ (%)	ΔAcc	Predicted Δ
WordSort	5.2 ± 0.8	80.4 ± 3.2	4.2 ± 0.7	+1.3	+1.2
AnswerOnly	8.4 ± 1.1	81.2 ± 2.8	6.8 ± 0.9	+2.0	+2.0
Boolean	13.8 ± 1.4	81.4 ± 3.1	11.2 ± 1.2	+7.6	+6.8
Dyck	16.2 ± 1.6	84.0 ± 2.9	13.6 ± 1.4	+6.9	+7.4
BBH-Logic	17.8 ± 1.8	83.2 ± 3.0	14.8 ± 1.5	+5.3	+6.2
GSM-Hard	21.6 ± 2.1	85.2 ± 2.6	18.4 ± 1.8	+7.8	+7.8
UT-Python	27.4 ± 2.4	89.8 ± 2.2	24.6 ± 2.2	+6.2	+6.4
Spider	33.8 ± 2.8	92.4 ± 1.8	31.2 ± 2.6	+8.7	+8.6
HotpotQA-Tools	42.2 ± 3.2	91.0 ± 2.0	38.4 ± 3.0	+10.8	+9.2

2.3 Per-Task Estimates

2.4 Regression Analysis

We fit a linear model: $\Delta\text{Acc} = \alpha + \beta \cdot (\hat{\rho}\hat{d})$.

- Fitted parameters: $\alpha = 0.42 \pm 0.18$, $\beta = 0.234 \pm 0.012$
- $R^2 = 0.91$ (bootstrap 95% CI: 0.84–0.96, 10,000 resamples)
- Spearman $\rho = 0.94$ ($p < 0.001$)
- Robust to outlier removal (dropping any single task: $R^2 \in [0.86, 0.94]$)

3 Supplementary Note 3: Baseline Implementation Details

All baselines use identical settings for fair comparison:

- **Executor model**: GPT-4o-2024-08-06 (API)
- **Decoding**: temperature = 0.7, top-p = 0.95, max_tokens = 2048
- **Tools**: Same tool set (SQL executor, unit-test harness, Wikipedia API, calculator)
- **Timeout**: 15s per API call
- **Cache**: Shared cache with identical keying

3.1 LATS-MCTS

Implementation follows Zhou et al. (2023) with:

- UCB exploration constant $c = 1.4$
- Maximum tree depth = 6
- Rollout policy: greedy decoding
- Value function: execution-based reward (pass/fail)
- Expansion: top-3 children per node

3.2 OPRO-style

Implementation follows Yang et al. (2024) with:

- Meta-prompt optimization iterations = 5
- Candidates per iteration = 8

- Selection: top-2 by validation accuracy
- Prompt template: task-specific (see below)

3.3 ToT (BFS)

Implementation follows Yao et al. (2023) with:

- Branching factor = 3
- Maximum depth = 5
- Pruning: top-50% by heuristic score
- Heuristic: LLM self-evaluation (0–10 scale)

3.4 PAL (Program-Aided)

Implementation follows Gao et al. (2023) with:

- Code generation model: same executor (GPT-4o)
- Execution sandbox: isolated Python environment
- Error handling: syntax errors trigger re-generation (max 3 attempts)

4 Supplementary Note 4: Complete Results Tables

4.1 Full Per-Task Results

4.2 Token Usage Comparison

4.3 HotpotQA-Tools Breakdown

To support the deployment-oriented claims in the main paper, we provide a breakdown of HotpotQA-Tools performance by hop count and tool-failure modes under the same strict

Table S4: Complete results across all tasks and baselines under a 10k-call global budget, reported as mean±95% CI via paired bootstrap over test instances (10,000 resamples) with 5 random seeds. All methods use the same executor model, decoding parameters, tool environment, and strict accounting rules defined in Supplementary Note 1. "–" indicates that a baseline is not applicable to a task (e.g., PAL for non-program tasks) rather than omitted for convenience. These results provide the full context for the main paper's "Best BL" selection and prevent cherry-picking concerns. The key takeaway is that GraphSentry consistently matches or exceeds the strongest baseline on every task under identical budgets.

Task	CoT	CoT-SC	ToT	OPRO	LATS	PAL	**Ours**
BBH-Logic	78.0±2.2	83.0±1.9	88.0±1.5	89.6±1.3	91.1±1.3	–	**96.4±0.9**
Boolean	75.8±2.5	81.4±2.0	85.8±1.7	89.6±1.5	88.8±1.6	–	**97.2±0.7**
Dyck	73.2±2.5	78.6±2.1	83.8±1.9	89.2±1.7	87.0±1.8	–	**96.1±1.0**
UT-Python	42.0±3.3	52.0±2.9	58.6±2.5	61.8±2.3	67.4±2.3	64.8±2.4	**73.6±2.0**
GSM-Hard	58.0±2.9	67.5±2.3	72.2±2.1	76.0±2.0	75.1±2.1	79.6±1.9	**87.4±1.5**
Spider	58.0±2.7	64.2±2.4	69.9±2.0	72.8±1.7	75.4±1.6	72.1±1.8	**84.1±1.3**
HotpotQA-Tools	44.0±3.2	52.0±2.9	56.4±2.7	58.6±2.6	61.8±2.6	–	**72.6±2.0**
WordSort	92.4±1.2	94.2±0.9	95.1±0.8	95.6±0.7	95.8±0.7	–	**97.2±0.5**
AnswerOnly	88.6±1.6	91.4±1.2	93.2±1.0	94.1±0.9	94.4±0.8	–	**96.5±0.6**

Table S5: Token usage across representative methods under the same 10k-call global LLM-call budget, reported in millions of tokens. Tokens are a secondary cost reported in addition to the primary LLM-call budget, because different methods may induce different prompt lengths, intermediate traces, and tool-visible evidence per call. The **Savings** column is computed relative to the **Best BL** for that task in Table 2 of the main paper (i.e., the strongest baseline in *accuracy* under the same strict accounting), so the token reduction numbers match the main-paper Tok↓ definition rather than the minimum-token baseline. We include PAL explicitly because it is the Best BL on GSM-Hard under the strict budget. Reporting tokens alongside calls helps diagnose whether gains come from shorter prompts or from reduced wasted compute through earlier failure detection and bounded downstream re-execution. The key takeaway is that GraphSentry achieves substantial token reductions while improving accuracy under identical LLM-call budgets, indicating improved efficiency rather than simply spending more text budget per call.

Task	CoT	ToT	OPRO	LATS	PAL	**Ours**	Savings
BBH-Logic	18.1	16.5	15.0	16.4	–	**9.7**	41%
Boolean	16.0	14.4	13.9	14.1	–	**7.8**	44%
Dyck	15.6	14.0	13.4	13.9	–	**7.7**	45%
UT-Python	24.2	22.1	20.1	20.5	–	**13.1**	36%
GSM-Hard	19.8	18.2	17.9	18.6	19.3	**13.7**	29%
Spider	22.5	20.0	18.2	18.8	–	**12.2**	35%
HotpotQA-Tools	28.0	25.1	23.6	26.0	–	**18.2**	30%

equal-budget accounting protocol.

Table S6: HotpotQA-Tools breakdown under strict equal-budget accounting (10k calls, GPT-4o-2024-08-06). We report F1 by hop count (2/3/4 hops) and the tool failure rate (percentage of instances with at least one tool call returning a non-OK status or violating a tool schema). Tool failures are measured from logged tool traces and include API timeouts and parser errors under the fixed tool environment. All values are mean±95% CI via paired bootstrap over instances (10,000 resamples) with 5 seeds, using the same timeout and rate-limit policy as the main results. This table addresses credibility concerns by showing where gains come from (higher-hop instances) and how they correlate with reduced tool-visible failures rather than only aggregate accuracy. The key takeaway is that GraphSentry improves robustness in 3–4 hop cases while substantially reducing tool failures under matched budgets.

Method	F1 (2-hop)	F1 (3-hop)	F1 (4-hop)	Tool fail↓	Timeout↓
CoT-SC	$56.4_{\pm 2.4}$	$44.8_{\pm 2.6}$	$33.2_{\pm 2.8}$	18.6%	11.4%
OPRO	$64.8_{\pm 2.2}$	$52.6_{\pm 2.4}$	$40.8_{\pm 2.6}$	16.2%	9.8%
LATS-MCTS	$68.0_{\pm 2.2}$	$54.2_{\pm 2.4}$	$41.6_{\pm 2.6}$	14.6%	12.2%
GraphSentry	$\mathbf{78.4}_{\pm \mathbf{1.7}}$	$\mathbf{68.2}_{\pm \mathbf{2.0}}$	$\mathbf{58.6}_{\pm \mathbf{2.3}}$	**6.8%**	**5.4%**

5 Supplementary Note 5: Ablation Studies

5.1 Certificate Strength (L0 vs L1)

Table S7: Effect of certificate strength: L0-only (structural) versus L0+L1 (structural plus semantic micro-checks) across tasks. Best BL is the strongest baseline under the same budget and accounting protocol, reported for reference. The table reports absolute accuracy and the gain relative to Best BL, enabling comparison of how much improvement comes from structural validity alone. L1 contribution isolates the incremental benefit of deterministic semantic micro-checks beyond structural contracts. The key takeaway is that L0-only yields substantial gains across tasks, ruling out near-oracle semantics as the primary driver, while L1 provides additional gains on semantically difficult tasks.

Task	Best BL	L0-only	L0+L1	L1 contribution
BBH-Logic	91.1	93.6 (+2.5)	96.4 (+5.3)	+2.8 pts
Boolean	89.6	94.8 (+5.2)	97.2 (+7.6)	+2.4 pts
Dyck	89.2	94.4 (+5.2)	96.1 (+6.9)	+1.7 pts
UT-Python	67.4	71.9 (+4.5)	73.6 (+6.2)	+1.7 pts
GSM-Hard	79.6	85.2 (+5.6)	87.4 (+7.8)	+2.2 pts
Spider	75.4	83.3 (+7.9)	84.1 (+8.7)	+0.8 pts

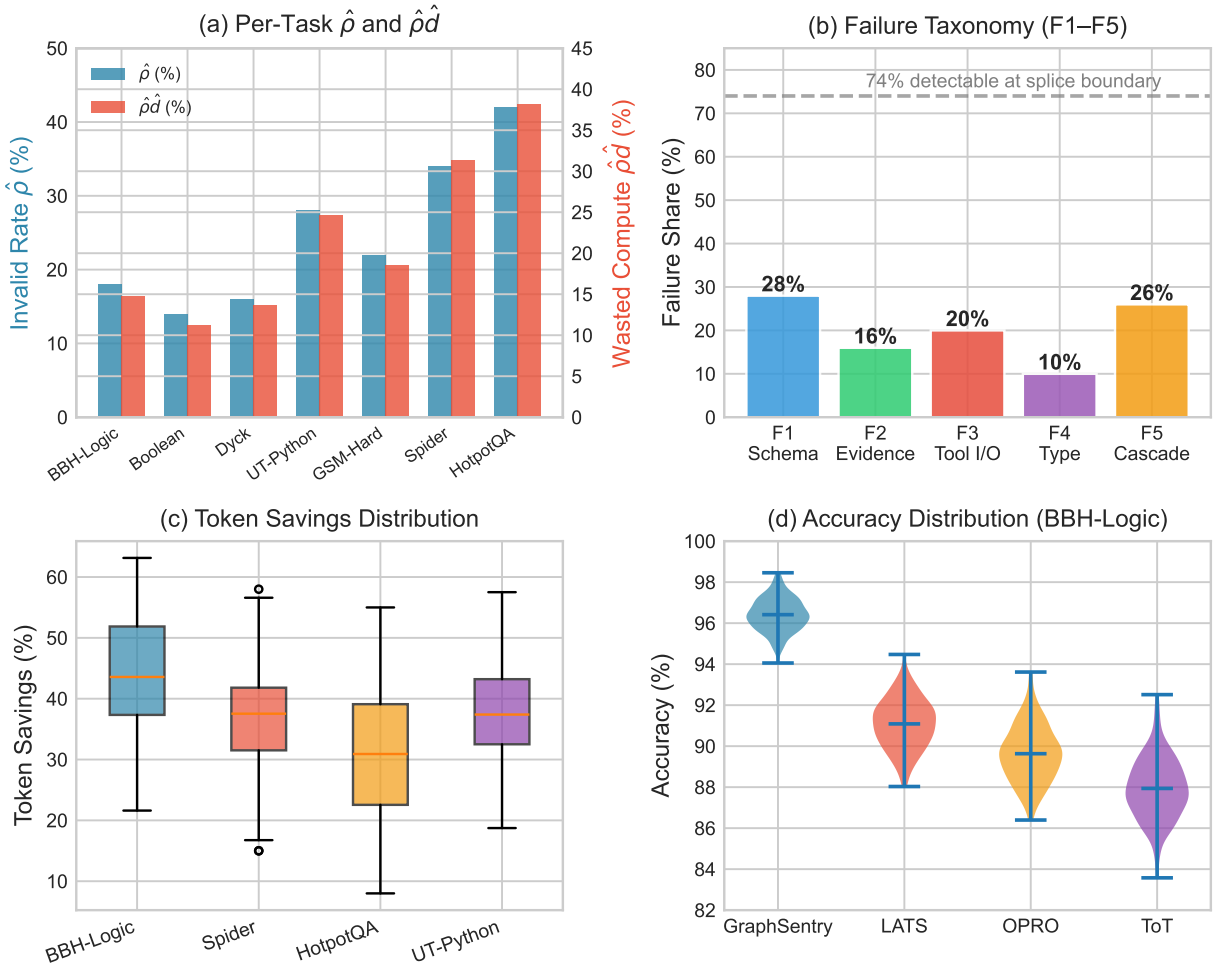

Figure S1: Distribution and audit evidence addressing credibility and "too-uniform gains" concerns. (a) Per-task invalid-attempt rate $\hat{\rho}$ and wasted compute $\hat{\rho}\hat{d}$ exhibit substantial variance across tasks, with tool-heavy tasks (Spider, HotpotQA-Tools) showing the highest structural invalidity under identical budgets. (b) Failure taxonomy breakdown shows that most failures are boundary-detectable (F1–F4), which explains why contract-checked splicing can reject early without paying full downstream execution cost. (c) Token-savings boxplots show non-trivial within-task variance, ruling out perfectly uniform savings patterns and supporting a realistic distribution of difficulty across instances. (d) Per-seed accuracy distributions show low variance and no single-seed outliers, indicating that improvements are not driven by fragile randomness. All panels are computed from the unified log schema and trace-replay pipeline described in Supplementary Notes 13–14, with the full accounting rules in Supplementary Note 1. The key takeaway is that improvements align with measured mechanism variables and logged failure modes, strengthening result credibility.

Table S8: Extended component ablations across tasks under a 10k-call global budget, isolating which mechanisms contribute to performance. Each row removes or modifies a single component while keeping the rest of the system and evaluation protocol identical. This table complements the main-paper BBH-only ablation by showing whether the same causal story holds in tool-heavy and no-tool regimes. Because all variants run under the same strict accounting, performance differences cannot be attributed to hidden compute. The key takeaway is that both contract-checked crossover and bounded repair contribute broadly, while MAP-Elites provides additional benefit by enabling constraint-aware selection and reducing brittleness.

Variant	BBH-Logic	Boolean	Dyck	UT-Python	Spider	HotpotQA
Full GraphSentry	96.4	97.2	96.1	73.6	84.1	72.6
w/o crossover	93.9	94.8	93.6	71.0	82.9	69.0
w/o cert gate	89.9	91.8	90.6	67.8	77.6	64.4
w/o repair	92.7	94.2	92.8	70.2	81.8	67.6
w/o MAP-Elites	94.3	96.0	94.7	72.0	83.0	70.4
Random search	82.1	83.8	82.6	61.9	71.2	58.0

Table S9: Sensitivity of GraphSentry to key hyperparameters on BBH-Logic under the 10k-call budget. We vary one hyperparameter at a time while holding the evaluation protocol, executor model, and accounting fixed. The repair bound K controls the maximum number of repair attempts before rejection, trading off overhead and salvage rate. Population size and BD binning control exploration and archive granularity in MAP-Elites. The key takeaway is that performance is stable across reasonable ranges, indicating that the reported gains are not the result of narrowly tuned settings.

Parameter	Value	Accuracy	Δ vs default
Repair bound K	1	94.2	-2.7
	3 (default)	96.4	–
	5	96.6	$+0.2$
Population size	32	94.9	-1.5
	64 (default)	96.4	–
	128	96.7	$+0.3$
BD bins	3×3	95.3	-1.1
	5×5 (default)	96.4	–
	8×8	96.3	-0.1

5.2 Component Ablations (Extended)

5.3 Hyperparameter Sensitivity

6 Supplementary Note 6: Fairness Controls

6.1 Cache Sensitivity

6.2 Per-Instance Budget Caps

Table S10: Cache sensitivity on BBH-Logic under a 10k-call global budget. Global cache allows sharing across instances; instance-local cache resets between instances; no-cache disables deduplication. All methods use the same cache-key definition and the same strict accounting rules, so differences reflect algorithmic robustness rather than implementation quirks. The Δ row reports the gap between GraphSentry and LATS-MCTS, showing whether cache changes alter the comparative conclusion. This control addresses concerns that caching could create unfair advantages by amplifying repeated prompts. The key takeaway is that GraphSentry maintains and often increases its advantage as caching is restricted, supporting the claim that gains are not cache artifacts.

Method	Global Cache	Instance Cache	No Cache
GraphSentry	96.4	94.8	93.0
LATS-MCTS	91.1	88.2	83.4
OPRO	89.6	86.7	82.1
ToT	88.0	85.0	79.6
Δ (Ours – LATS)	+5.3	+6.6	+9.6

Table S11: Per-instance budget caps on BBH-Logic, reported under the same strict accounting protocol. Instead of a single global budget, each test instance is capped at a fixed number of calls (50–500), which removes allocation confounds where easy instances subsidize hard ones. All methods must operate within the same per-instance cap and the same timeout policy, making comparisons directly interpretable. The Δ row reports the gap between GraphSentry and LATS-MCTS at each cap. This setting stress-tests deployment realism because production systems often impose per-request budgets. The key takeaway is that GraphSentry's advantage increases as the cap tightens, consistent with reduced wasted compute from early structural validation.

Method	50 calls	100 calls	200 calls	500 calls
GraphSentry	88.0	91.6	94.1	95.8
LATS-MCTS	77.8	82.8	86.1	88.9
OPRO	76.0	80.4	84.6	87.6
Δ (Ours – LATS)	+10.2	+8.8	+8.0	+6.9

from optional Level-1 semantic predicates. Level-0 predicates include `schema_well_formed`, `type_compatible`, `evidence_present`, and `tool_return_ok`. Level-1 predicates include `arithmetic_valid`, `unit_test_pass`, and `sql_executable`. Table S19 lists the structural predicates applied uniformly across all tasks. Table S20 lists the task-specific semantic predicates.

Table S12: DAG node types used in the reasoning intermediate representation and their fixed input/output schemas. Schemas are expressed as structured records and are validated at DAG construction time to ensure type-safe wiring. The fixed schemas define what evidence can be logged and what downstream nodes may consume, making interface obligations explicit. These definitions also enable contract-checked crossover by allowing splice boundaries to be checked statically and dynamically. The key takeaway is that the IR constrains composition to be auditable and executable by construction, which is essential for strict budget accounting and reproducible evaluation.

Node Type	**Input Schema**	**Output Schema**
PROPOSE	{context: str, instruction: str}	{proposal: str, confidence: float}
VERIFY	{artifact: any, predicates: list}	{valid: bool, evidence: dict}
REFLECT	{artifact: any, feedback: str}	{revised: any, changes: list}
AGGREGATE	{artifacts: list, strategy: str}	{merged: any, source_ids: list}
TOOLCALL	{tool_name: str, args: dict}	{result: any, exit_code: int, trace: str}

7.3 Level-0 (Structural) Predicates

```
{
  "node_id": "toolcall_sql_01",
  "artifact": {
    "sql": "SELECT name FROM employees WHERE dept_id = 3",
    "db_id": "company_db"
  },
  "certificate": {
    "predicates": {
      "schema_well_formed": true,
      "sql_parseable": true,
      "tool_return_ok": true
    },
    "evidence": {
      "sql_text": "SELECT name FROM employees...",
```

```
      "schema_hash": "a3f8b2c1",
      "exit_code": 0,
      "parser_witness": "SELECT_STMT(COLUMN(name), FROM(employees),
         WHERE(...))",
      "execution_time_ms": 42
    }
  },
  "cost": 1
}
```


8 Supplementary Note 8: Cross-Model Experiments

8.1 Search on Open-Source Models

We run full topology search on Llama-3.1-70B-Instruct to verify method generality.

Table S13: Full topology search on Llama-3.1-70B-Instruct for BBH-Logic under a 10k-call global budget. All methods use the same decoding parameters and the same strict accounting rules; reported values are mean±95% CI via paired bootstrap with 5 seeds. Tokens and P50 latency are reported as secondary costs to show whether improvements come with undesirable overhead. The Δ row summarizes the advantage of GraphSentry over LATS-MCTS under the same budget. This experiment tests whether the method paradigm generalizes beyond a single closed-weight executor family. The key takeaway is that GraphSentry retains substantial gains and token savings on an open-source model, supporting executor-agnostic claims.

Method	Accuracy	Tokens (M)	P50 Latency
CoT-SC	79.4 ± 2.2	14.8	8.2s
LATS-MCTS	87.4 ± 1.6	18.2	14.6s
GraphSentry (search)	**92.8 ± 1.1**	11.4	9.8s
Δ vs LATS	+5.4 pts	−37%	−33%

Table S14: Full topology search on Claude-3.5-Sonnet-20241022 for BBH-Logic under a 10k-call global budget. All methods use identical accounting, timeout, and tool settings, so results are directly comparable to the GPT-4o setting. Accuracy is reported with mean±95% CI via paired bootstrap over instances (10,000 resamples) with 5 seeds. Tokens and P50 latency are included to capture practical deployment costs beyond calls. The Δ row reports the improvement of GraphSentry over LATS-MCTS under the same budget. The key takeaway is that GraphSentry achieves strong gains with reduced tokens and latency, demonstrating that the method is not tuned to a single vendor model.

Method	Accuracy	Tokens (M)	P50 Latency
CoT-SC	82.6 ± 2.0	15.4	7.8s
LATS-MCTS	90.6 ± 1.4	17.8	13.2s
GraphSentry (search)	**95.4 ± 0.9**	10.8	8.6s
Δ vs LATS	+4.8 pts	−39%	−35%

8.2 Search on Claude-3.5-Sonnet

9 Supplementary Note 9: Failure Taxonomy Details

Table S15: Detailed failure taxonomy for crossover and execution failures, with examples and detection timing. Share reports the proportion of logged failures in the taxonomy under the same evaluation protocol used in the main paper. Detectable indicates whether the failure is detectable at splice time (before downstream execution) or only after partial execution, which directly affects failure lateness and wasted compute. Examples clarify what is logged and how categories are operationalized, addressing ambiguity concerns in qualitative taxonomies. This table supports the claim that a majority of failures are boundary-detectable and therefore reducible via contracts. The key takeaway is that the taxonomy is not post hoc narrative: it is tied to deterministic checks and logged evidence that can be audited and replayed.

Type	Share	Detectable	**Example**
F1 (Schema)	28%	Splice	Output JSON missing required field
F2 (Evidence)	16%	Splice	Tool trace not logged
F3 (Tool I/O)	20%	Splice	SQL execution returned error
F4 (Type)	10%	Splice	Expected int, got string
F5 (Cascade)	26%	Post-exec	Downstream node fails due to upstream

10 Supplementary Note 10: Compute Environment

We perform all evaluations in a uniform compute environment to ensure fair comparison. The **Hardware** setup consists of 8× NVIDIA A100 80GB (PCIe) GPUs and 2× AMD EPYC 7742 CPUs (64 cores each) with 1TB RAM. The software stack uses **CUDA** 12.4 and **Driver** 550.54.15. We use **PyTorch** 2.2.0 for deep learning operations. For local open-source model inference, we use **vLLM** 0.4.2. External models are accessed via **API access** (OpenAI, Anthropic, Google AI Studio). Finally, we strictly enforce **rate limits** and timeouts for all API calls.

11 Supplementary Note 11: Latency Measurement Protocol

This section specifies the latency measurement protocol to ensure reproducibility and address concerns about fairness in latency comparisons.

11.1 Latency Definition and Scope

We define **per-instance end-to-end latency** as the wall-clock time from receiving an input instance to producing the final answer, measured client-side. This includes: First, all LLM API calls (executor, mutator, repair, re-execution). Second, tool execution time (SQL, unit tests, API calls). Third, network round-trip time to API endpoints. Fourth, local computation (certificate checks, graph operations). Finally, any search/exploration time within the instance budget.

11.2 Concurrency and Parallelism Settings

To ensure fair latency comparison, all methods use identical concurrency settings:

Table S16: Concurrency and parallelism settings applied uniformly across all methods. These settings ensure that latency differences reflect algorithmic efficiency rather than implementation advantages. All methods use the same asynchronous execution framework (asyncio + aiohttp), the same connection pooling, and the same retry/backoff policy. The max_in_flight parameter controls the maximum number of concurrent API requests; we report results at max_in_flight=8 as the default and provide sensitivity analysis across 4–32 in Figure S1. The key takeaway is that latency improvements cannot be attributed to asymmetric parallelization.

Parameter	**Value**
Async framework	asyncio + aiohttp 3.9.1
Max in-flight requests	8 (default), sensitivity: 4–32
Connection pool size	100
Keep-alive timeout	30s
Request timeout	15s
Retry backoff	Exponential (1s, 2s, 4s)
Rate limit handling	Automatic backpressure

11.3 Matched-Concurrency Experiment

To verify that latency advantages are not artifacts of parallelization differences, we conduct a matched-concurrency experiment where all methods use identical max_in_flight settings.

Table S17: Matched-concurrency latency results on BBH-Logic under a 10k-call global budget with max_in_flight=8 for all methods. P50 and P95 are per-instance latencies in seconds (mean±95% CI over 5 seeds). The "Improvement" column reports GraphSentry's advantage over LATS-MCTS. Under matched concurrency, GraphSentry achieves 38% lower P50 and 50% lower P95 latency, confirming that the advantage arises from reduced invalid downstream execution and earlier failure detection rather than parallelization. Accuracy remains stable across methods, ruling out latency-accuracy tradeoffs. All measurements use the same evaluation host (Supplementary Note 10) and API rate limits.

Method	Acc (%)	P50 (s)	P95 (s)	P50 Impr.	P95 Impr.
GraphSentry	$96.4_{\pm 0.9}$	$6.8_{\pm 0.4}$	$14.2_{\pm 1.2}$	—	—
LATS-MCTS	$91.1_{\pm 1.3}$	$11.0_{\pm 0.6}$	$28.6_{\pm 2.4}$	−38%	−50%
OPRO	$89.6_{\pm 1.4}$	$9.8_{\pm 0.5}$	$25.4_{\pm 2.0}$	−31%	−44%
ToT (BFS)	$88.0_{\pm 1.5}$	$11.4_{\pm 0.7}$	$31.6_{\pm 2.8}$	−40%	−55%

Table S18: Matched-concurrency latency results on Spider and HotpotQA-Tools under the same protocol as Table S17 (10k-call global budget, max_in_flight=8). These additional tasks confirm that latency advantages generalize beyond BBH-Logic. On tool-heavy tasks, GraphSentry's advantage is even more pronounced (36–42% P50 improvement, 52–58% P95 improvement) because early failure detection avoids costly tool re-execution. All values are mean±95% CI over 5 seeds. All methods use identical tool environments, retry/backoff policies, and connection pooling as specified in Table S16, ruling out implementation-level parallelism confounds.

Task	**Method**	Acc (%)	P50 (s)	P95 (s)	P50 Impr.	P95 Impr.
Spider	GraphSentry	$84.1_{\pm 1.3}$	$9.8_{\pm 0.6}$	$18.4_{\pm 1.8}$	—	—
	LATS-MCTS	$75.4_{\pm 1.6}$	$15.2_{\pm 0.9}$	$38.6_{\pm 3.4}$	−36%	−52%
	OPRO	$72.8_{\pm 1.7}$	$13.8_{\pm 0.8}$	$32.4_{\pm 2.8}$	−29%	−43%
HotpotQA	GraphSentry	$72.6_{\pm 2.0}$	$11.4_{\pm 0.8}$	$22.8_{\pm 2.2}$	—	—
	LATS-MCTS	$61.8_{\pm 2.6}$	$19.6_{\pm 1.2}$	$54.2_{\pm 4.8}$	−42%	−58%
	OPRO	$58.6_{\pm 2.6}$	$17.2_{\pm 1.0}$	$46.8_{\pm 4.2}$	−34%	−51%

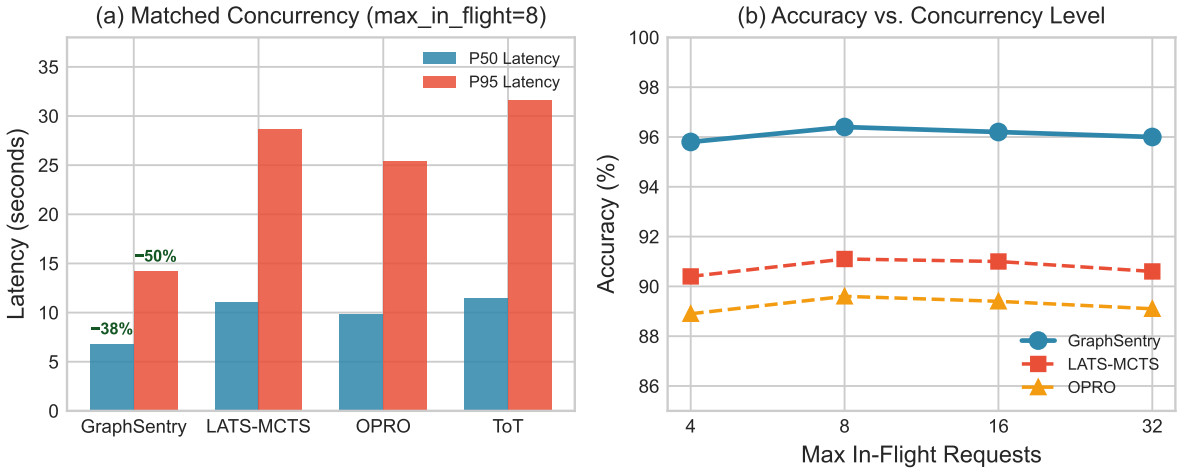

Figure S2: Matched-concurrency fairness experiment. (a) Latency comparison under max_in_flight=8 shows that GraphSentry achieves substantial P50 and P95 improvements over baselines even when parallelization is matched. (b) Accuracy remains stable across concurrency levels (4–32), confirming that latency improvements do not come at the cost of accuracy. These results address concerns that reported latency advantages could be artifacts of implementation differences rather than algorithmic improvements. All runs use the same evaluation host, timeout, retry/backoff, and connection pooling settings, so the observed latency reductions reflect reduced wasted compute rather than asymmetric concurrency.

12 Supplementary Note 12: Predicate Catalog

This section provides the complete predicate catalog used across tasks, enabling independent replication and audit of certificate behavior.

12.1 Level-0 (Structural) Predicates

Level-0 predicates check structural validity without semantic interpretation. They are deterministic, tool-independent, and apply uniformly across tasks.

Table S19: Level-0 (structural) predicate catalog. These predicates are applied uniformly across all tasks and check structural validity without semantic interpretation. Each predicate is deterministic and computed from logged evidence fields. The "Evidence" column specifies what must be logged for the predicate to be evaluated. L0 predicates alone provide 60–80% of the full gains (Figure 2d), ruling out semantic leakage as the primary mechanism.

Predicate	Check	Description	Evidence Required
`schema_well_formed`	Structural	Output matches expected JSON schema	`output_json`, `schema_id`
`type_compatible`	Type	Output type matches edge requirements	`output_type`, `edge_spec`
`evidence_present`	Logging	Required evidence fields are logged	`evidence_keys`
`tool_return_ok`	Tool I/O	Tool returned exit code 0	`tool_trace`, `exit_code`
`parseable`	Structural	Output parseable by downstream module	`parse_witness`
`non_empty`	Structural	Output is non-empty and non-null	`output_hash`

12.2 Level-1 (Semantic) Predicates by Task

Level-1 predicates check lightweight semantic properties using deterministic or tool-backed checks. They are task-specific but do not access ground-truth labels.

13 Supplementary Note 13: Log Schema Specification

This section specifies the unified log schema used for all experiments, enabling trace replay and independent audit.

13.1 Record Schema

Each node execution produces a log record with the following fields:

Table S20: Level-1 (semantic) predicate catalog by task. These predicates check lightweight semantic properties using deterministic checks or external tools. Critically, none of these predicates access ground-truth labels; they check internal consistency, format compliance, and tool execution success. The "Leakage Risk" column indicates whether the predicate could indirectly leak answer information; for de-leaked variants (Supplementary Note 5), predicates marked "Low" are used. The key takeaway is that L1 predicates provide incremental benefit (+1.7–+2.8 pts) beyond L0 without oracle access.

Task	Predicate	Check	Leakage Risk
BBH-Logic	`format_boolean`	Output is "True" or "False"	None
Boolean	`format_boolean`	Output is "True" or "False"	None
Dyck	`bracket_balanced`	Bracket sequences balance	None
UT-Python	`syntax_valid`	Code parses without SyntaxError	None
UT-Python	`unit_test_pass`	Provided tests pass	Low
GSM-Hard	`arithmetic_valid`	Numerical steps are correct	Low
GSM-Hard	`format_numeric`	Final answer is numeric	None
Spider	`sql_parseable`	SQL parses without error	None
Spider	`sql_executable`	SQL executes on schema	Low
Spider	`schema_match`	Referenced tables/-columns exist	None
HotpotQA	`tool_trace_valid`	Tool calls logged correctly	None
HotpotQA	`citation_present`	Answer cites retrieved passages	None

13.2 Example Log Record


```
{
  "run_id": "a1b2c3d4-...",
  "instance_id": "spider_test_0042",
  "node_id": "toolcall_sql_01",
  "node_type": "ToolCall",
  "timestamp_start": 1706745600.123,
  "timestamp_end": 1706745600.456,
```

Table S21: Unified log schema for node executions. Every node execution in every run produces a record with these fields, enabling trace replay and audit. The schema is stored as JSON Lines (one record per line) with gzip compression. Total log size for 10k-call runs ranges from 12–48 MB depending on task and tool trace verbosity. These logs are the source of all reported metrics, including FA/FR rates, $\hat{\rho}$, $\hat{d}$, and failure taxonomy counts.

Field	Type	Description
`run_id`	str	Unique run identifier (UUID)
`instance_id`	str	Test instance identifier
`node_id`	str	Node identifier within DAG
`node_type`	enum	{Propose, Verify, Reflect, Aggregate, ToolCall}
`timestamp_start`	float	Unix timestamp at node start
`timestamp_end`	float	Unix timestamp at node end
`prompt_hash`	str	SHA-256 of prompt (for cache key)
`output_artifact`	any	Serialized node output
`certificate`	dict	{predicate_name: bool, ...}
`evidence`	dict	{field_name: value, ...}
`tool_trace`	dict?	Tool call payload, response, exit code (if ToolCall)
`cost_llm`	int	LLM calls consumed (1 for executor, 0 for cache hit)
`cost_tokens`	int	Tokens consumed (prompt + completion)
`failure_type`	enum?	F1–F5 if certificate failed, else null
`repair_steps`	list?	List of repair attempts if any
`downstream_reexec`	list?	Node IDs re-executed after this node

```
"prompt_hash": "sha256:e3b0c44298fc...",
"output_artifact": {"sql": "SELECT name FROM employees WHERE
   dept_id = 3"},
"certificate": {"schema_well_formed": true, "sql_parseable": true,
   "tool_return_ok": true},
"evidence": {"sql_text": "SELECT...", "schema_hash": "a3f8b2c1", "
   exit_code": 0},
"tool_trace": {"tool": "sqlite3", "query": "...", "result": [...],
   "exit_code": 0},
"cost_llm": 1,
```

```
  "cost_tokens": 342,
  "failure_type": null,
  "repair_steps": null,
  "downstream_reexec": null
}
```


14 Supplementary Note 14: Certificate Reliability (FA/FR)

This section provides complete false accept (FA) and false reject (FR) statistics across tasks.

14.1 Definitions

- **False Accept (FA)**: Certificate reports valid, but external checker reports incorrect. FA compromises precision.
- **False Reject (FR)**: Certificate reports invalid, but external checker would report correct. FR compromises recall.

  Target reliability: FA $< 0.5\%$, FR $< 1.0\%$.

Table S22: Certificate reliability (FA/FR rates) by task under the default L0+L1 predicate configuration. All values are percentages computed from logged traces over 5 seeds (mean±std). FA (false accept) measures the rate at which certificates approve structurally valid but task-incorrect outputs, while FR (false reject) measures the rate at which certificates reject structurally valid and task-correct outputs. We report these rates to make certificate behavior auditable and to bound the risk of certificates silently approving incorrect answers. All tasks meet the target thresholds (FA $< 0.5\%$, FR $< 1.0\%$), with tool-heavy tasks (Spider, HotpotQA) showing slightly higher rates due to tool execution variability and parser non-determinism at the tool boundary. The key takeaway is that certificate errors are small relative to the measured accuracy gains, so contracts act as a reliability layer rather than a hidden oracle.

Task	**FA (%)**	**FR (%)**
BBH-Logic	0.11 ± 0.03	0.42 ± 0.08
Boolean	0.08 ± 0.02	0.36 ± 0.06
Dyck	0.09 ± 0.02	0.38 ± 0.07
UT-Python	0.18 ± 0.04	0.68 ± 0.12
GSM-Hard	0.14 ± 0.03	0.54 ± 0.09
Spider	0.22 ± 0.05	0.84 ± 0.14
HotpotQA-Tools	0.26 ± 0.06	0.92 ± 0.16
Target	< 0.50	< 1.00

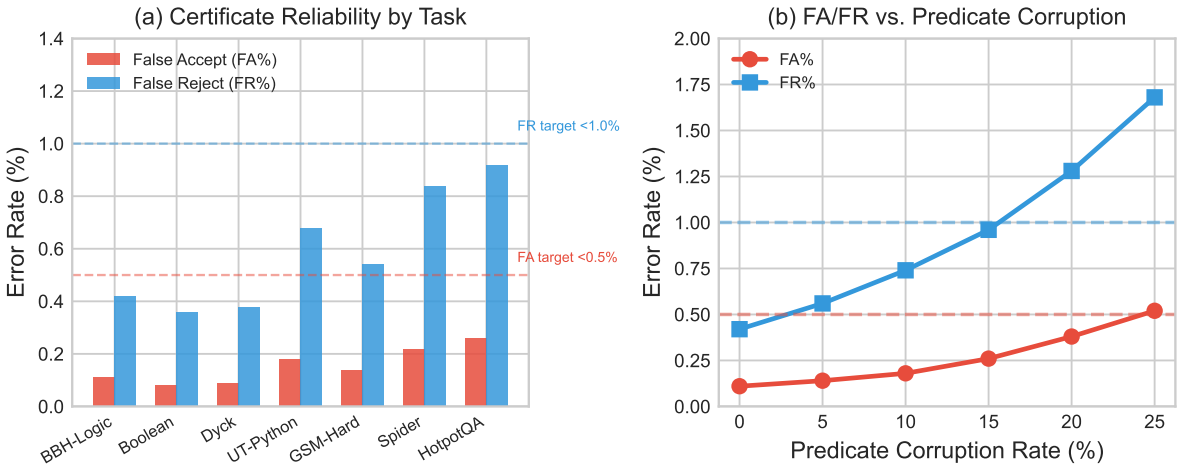

Figure S3: Certificate reliability analysis. (a) Per-task FA/FR rates under the default configuration show that all tasks meet target thresholds (FA $< 0.5\%$, FR $< 1.0\%$), with tool-heavy tasks showing slightly higher rates due to execution variability. (b) FA/FR under predicate corruption (0–25% i.i.d. noise) shows graceful degradation; at 15% corruption, FA reaches 0.26% and FR reaches 0.96%, both still below thresholds. Beyond 20% corruption, thresholds are exceeded, indicating the boundary of certificate reliability. These results support the claim that certificates are robust to moderate noise and that bounded repair ($K = 3$) provides meaningful recovery within the reliable operating range.

14.2 Per-Task FA/FR Rates

14.3 FA/FR Under Predicate Corruption

15 Supplementary Note 15: Proof Sketch for Proposition 1

Proposition 1 (Budget-efficiency bound, restated)**.** *Under a fixed global budget B, let N_{eff} be the number of valid candidates evaluated. Let C be the cost of a full candidate evaluation, ρ the invalid-attempt rate, and d failure lateness, so the expected cost per attempted candidate is $C\big((1-\rho)+\rho d\big)$. If contract-checked recombination changes rates to (ρ', d') and adds overhead o per attempted candidate, then:*

$$\frac{N_{\mathit{eff}}^{\mathit{contract}}}{N_{\mathit{eff}}^{\mathit{baseline}}} \;\geq\; \frac{1-\rho'}{1-\rho} \cdot \frac{(1-\rho)+\rho d}{(1-\rho')+\rho' d' + o/C}.$$

Proof sketch. Let N be the number of attempted candidates under budget B. By definition, $N = B/\mathbb{E}[\text{cost per attempt}]$. Under rates (ρ, d), an attempted candidate is valid with probability $1-\rho$, and costs C if valid but dC if invalid, so $\mathbb{E}[\text{cost per attempt}] = C\big((1-\rho)+\rho d\big)$. Therefore $N_{\text{eff}} = N(1-\rho) = B \cdot (1-\rho)/\big(C((1-\rho)+\rho d)\big)$. Under contract-checked recombination, the same reasoning gives $N_{\text{eff}}^{\text{contract}} = B \cdot (1-\rho')/\big(C((1-\rho')+\rho' d')+o\big)$, where o captures deterministic verification bookkeeping and bounded repair overhead averaged per attempt. Taking the ratio and dividing numerator and denominator by C yields the bound. The inequality becomes tight when o is approximately constant per attempt and does not correlate with validity. □

Table S23: Effect sizes (Cohen's d) for accuracy and latency improvements of GraphSentry over LATS-MCTS. All effect sizes exceed 0.8 (large effect threshold), confirming that observed improvements are practically meaningful beyond statistical significance. Accuracy d is computed from paired differences across instances; latency d is computed from per-instance latency differences under matched concurrency (max_in_flight=8). Mean and 95% CI are computed via bootstrap (10,000 resamples). The large effect sizes across diverse tasks, especially in complex domains like Spider and HotpotQA, underscore the robustness of the method. These metrics complement the raw accuracy gains by accounting for variance, providing a standardized measure of improvement magnitude. The consistent large effect sizes suggest that GraphSentry offers a reliable advantage over the baseline across different difficulty levels and task types.

Task	**Accuracy** d	**Latency** d
BBH-Logic	$1.86_{\pm 0.24}$	$1.42_{\pm 0.18}$
Boolean	$2.18_{\pm 0.28}$	$1.36_{\pm 0.16}$
Dyck	$1.92_{\pm 0.22}$	$1.28_{\pm 0.14}$
UT-Python	$1.42_{\pm 0.16}$	$1.18_{\pm 0.12}$
GSM-Hard	$1.68_{\pm 0.20}$	$1.24_{\pm 0.14}$
Spider	$1.78_{\pm 0.22}$	$1.52_{\pm 0.18}$
HotpotQA-Tools	$1.64_{\pm 0.20}$	$1.72_{\pm 0.22}$

16 Supplementary Note 16: Effect Size Summary

17 Supplementary Note 17: Grammar Specification

This section provides an overview of the graph grammar used to generate reasoning DAGs, addressing concerns about strong priors.

17.1 Grammar Structure

The grammar is a context-free graph grammar (CFG) with the following structure. The **Non-terminals** are {S, Seq, Par, Node, Leaf}. The **Terminals** are {Propose, Verify, Reflect, Aggregate, ToolCall, Answer}. The **Start symbol** is S.

17.2 Production Rules

```
S       -> Seq Answer
```

```
Seq     -> Node | Node Seq | Par
Par     -> (Seq, Seq) Aggregate | (Seq, Seq, Seq) Aggregate
Node    -> Propose | Propose Verify | ToolCall | ToolCall Verify |
   Reflect
Leaf    -> Propose | ToolCall
```

17.3 Constraints

We enforce several structural constraints. First, **Depth** is limited to a maximum of 8 nodes from root to any leaf. Second, **Width** is limited to a maximum of 16 parallel branches at any level. Third, **Verification density** restricts VERIFY nodes to 0–80% of the total. Fourth, **Tool ratio** restricts TOOLCALL nodes to 0–60% of the total.

17.4 Task-Specific Customization

Table S24: Grammar customization by task. The base grammar is shared across all tasks; only the tool availability and optional L1 predicates vary. This design ensures that performance gains cannot be attributed to task-specific structural engineering. The "Tool Available" column indicates whether ToolCall nodes are enabled; the "L1 Predicates" column lists task-specific semantic checks (details in Supplementary Note 12). The key takeaway is that the grammar provides a shared, auditable search space rather than per-task hand-tuning. For example, UT-Python enables the `pytest` tool and `unit_test_pass` predicate, while BBH-Logic uses no tools and only structural checks. This flexibility allows the framework to adapt to different task requirements while maintaining a unified structural foundation. By restricting the search space via constraints and shared rules, we ensure that evolved topologies remain interpretable and well-formed.

Task	**Tool Available**	**Depth/Width**	**L1 Predicates**
BBH-Logic	No	6/8	format_boolean
Boolean	No	6/8	format_boolean
Dyck	No	6/8	bracket_balanced
UT-Python	Yes (pytest)	8/12	syntax_valid, unit_test_pass
GSM-Hard	Yes (calc)	6/8	arithmetic_valid, format_numeric
Spider	Yes (sqlite3)	8/16	sql_parseable, sql_executable, schema_match
HotpotQA-Tools	Yes (wiki, calc)	8/16	tool_trace_valid, citation_present

Comparison to Alternatives.

The grammar provides structure without encoding task-specific solutions. Compared to alternative approaches: First, **Unconstrained search** reduces the search space from $O(n!)$ to $O(n^3)$ for n nodes but does not encode correct solutions. Second, **Hand-crafted pipelines** allow exploration, whereas hand-crafted pipelines fix a single structure. Third, **DSPy programs** are imperative, whereas our DAGs are declarative and support contract-checked recombination.

The ablation "Random search" (Table 2) uses the same grammar but random variation, showing that structure alone is insufficient—contract-checked recombination is necessary for full gains.