BNqMark: Bayesian-Network Inference in Large Language Models

Fernando Rodriguez FERNANDO.RODRIGUEZ@AILYLABS.COM
Aily Labs
Madrid, Spain

Bojan Mihaljević B.MIHALJEVIC@UPM.ES
Departamento de Inteligencia Artificial
Universidad Politécnica de Madrid
Boadilla del Monte, 28660, Madrid, Spain

Editors: Gustau Camps-Valls, Manuele Leonelli and Gherardo Varando

Abstract

We present the first systematic evaluation of exact Bayesian-network inference in frontier reasoning large language models (LLMs). We introduce BNqMark, a benchmark-generation framework for exact inference in discrete Bayesian networks, and evaluate BNqMark-20, a benchmark instance comprising 78 procedurally generated binary networks and 434 conditional probability queries with exact ground truth. Across 9 frontier reasoning LLMs and 2 evaluation protocols, we find that current models can solve many exact inference queries on small and medium Bayesian networks, especially by externalizing inference as code. Several models correctly answered queries whose exact solution would require more than 13,000 arithmetic operations under variable elimination, rising to 68,000 in the code-generation protocol. Model capability degrades as inference complexity increases: graph and query complexity are both useful predictors of success after controlling for prompt length, network size, and model. These results position Bayesian networks as a controlled testbed for studying probabilistic reasoning in LLMs and motivate future work on richer and more challenging probabilistic inference settings.

Keywords: Bayesian networks, probabilistic inference, large language models, benchmark

1. Introduction

Large language models (LLMs) are increasingly used for tasks that require reasoning under uncertainty. In many real-world settings, relevant information is incomplete and/or noisy, and decisions depend not only on retrieving facts, but also on manipulating (conditional) probabilities. Examples include diagnosing system failures from partial sensor readings, estimating delays from correlated upstream events, or revising beliefs as new evidence arrives. This raises a question: to what extent can current LLMs perform exact probabilistic inference?

As a first step towards answering this question, we consider an idealized setting in which the probabilistic model is fully specified, the correct answer is known exactly, and problem difficulty can be varied in principled ways. Bayesian networks (BNs) provide precisely this setting (Pearl, 1988; Koller and Friedman, 2009). They are a canonical formalism for reasoning under uncertainty and allow us to quantify the computational complexity of a given query. By presenting models directly with serialized conditional probability tables

© 2026 F. Rodriguez & B. Mihaljević.

(CPTs), we remove most natural-language ambiguity and isolate the computational core of the task: given a complete probabilistic model, can an LLM correctly answer a conditional inference query? Since we provide only format-level instructions, with no worked examples or BN-specific inference guidance, success requires the LLM to infer the BN structure from the serialized CPTs and carry out the necessary arithmetic. This setting therefore lets us gauge how well LLMs understand BNs.

Among prior work on probabilistic reasoning with LLMs, Nafar et al. (2025) is the closest to ours because it also uses BNs. Our setting differs in 3 important ways. First, we present the BN directly through serialized CPTs, which isolates exact inference from natural-language interpretation. Second, we design the benchmark to analyze sources of inference difficulty, including network size, treewidth, query total factor size (see Section 4.1), and prompt length. Third, we evaluate larger and more complex networks.

To carry out this study, we developed BNqMark[1], a benchmark-generation framework for exact conditional inference in discrete BNs. BNqMark can generate networks and queries at different levels of difficulty by varying network properties such as the number of variables, treewidth, variable arity, and CPT concentration or determinism. It can also vary query properties, including the number of target and evidence variables, their distance in the graph, and how much the evidence changes the target probability.

We used BNqMark to create BNqMark-20[2], the benchmark instance evaluated in this paper. BNqMark-20 contains 78 generated BNs with sizes ranging from 4 to 20 variables and treewidths up to 12. All variables are binary, and CPTs are sampled to ensure numerical diversity. From these networks, we derive 434 conditional probability queries of the form $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$, each paired with exact ground truth. The benchmark is generated procedurally, enabling controlled variation in network size, treewidth, CPT characteristics, and query complexity. To our knowledge, it supports the first systematic evaluation of exact BN inference across multiple frontier reasoning LLMs.

We evaluate LLMs under 2 complementary protocols. In a *raw reasoning* protocol, models must compute the answer directly from the serialized CPTs without external tools. In a *code generation* protocol, models are asked to write Python code that solves the same inference problem, with access to BN libraries such as `pgmpy` (Ankan and Textor, 2024) and `pyAgrum` (Ducamp et al., 2020). These protocols probe different capabilities: the first tests whether a model can internally carry out exact BN inference, while the second tests whether it can externalize the computation as reliable executable code.

Our experiments with 9 frontier reasoning LLMs show that current models can solve many exact inference queries on small and medium BNs, especially when they externalize the computation as code. Without tools, raw reasoning is reliable mainly on simpler queries. At the same time, this capability remains bounded by the complexity of the inference problem: across both protocols, accuracy degrades as treewidth and query complexity increase, and regression analyses show that both effects remain after controlling for prompt length, network size, and model identity. Together, these findings indicate that BNqMark provides a useful controlled setting for analyzing probabilistic inference in LLMs and that task difficulty tracks both graph-level and query-level complexity.

1. https://github.com/ferjorosa/bnqmark
2. https://huggingface.co/datasets/ferjorosa/bnqmark-20

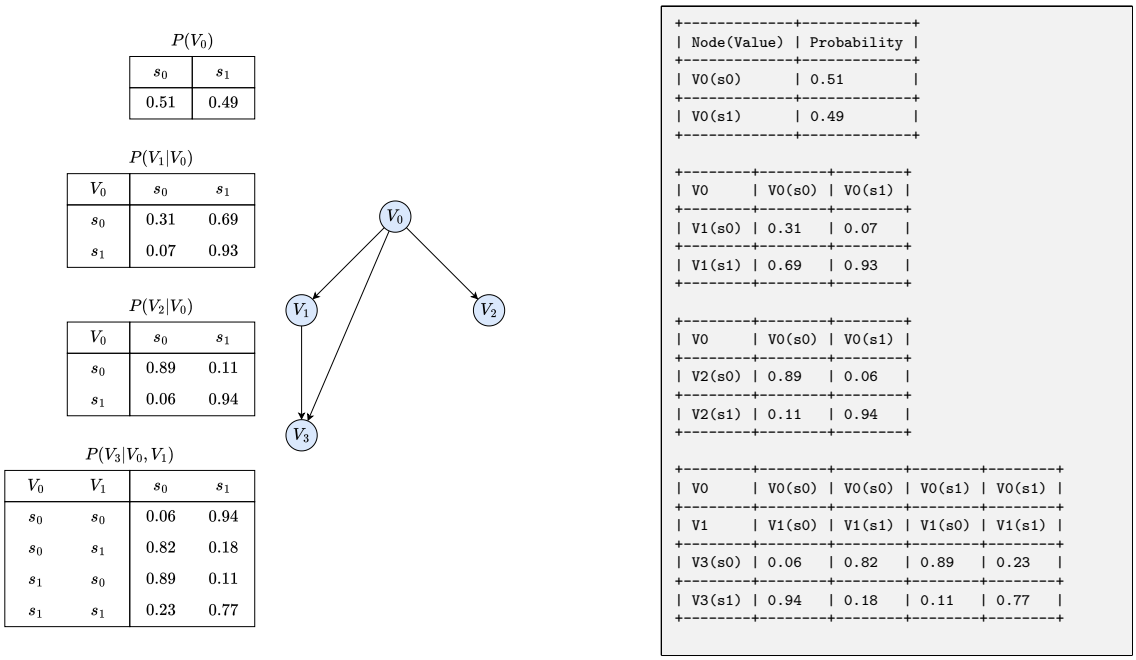
The main contributions of this paper are twofold: a systematic analysis of BN inference in frontier LLMs and a framework for generating benchmark BN and query datasets at specified levels of complexity for future studies.

2. Related Work

Probabilistic reasoning with language models. Recent work has shown growing interest in whether LLMs can reason quantitatively about uncertainty. Much of the broader reasoning-benchmark literature relies on natural-language tasks, which conflate linguistic interpretation with computation. More recent structural benchmarks such as ZebraLogic (Lin et al., 2025) instead vary difficulty through formal properties of the task. In the probabilistic setting, Paruchuri et al. (2024) study whether LLMs can estimate probabilities and sample from univariate distributions, while Qiu et al. (2026) examine probabilistic belief formation and updating in a broader Bayesian setting. Methodologically, Nafar et al. (2025) is the closest prior work because it also evaluates LLMs on BN reasoning tasks. Their setting, however, focuses on reasoning over uncertain text: the model must interpret a natural-language description and map it to an underlying BN before answering a query. In contrast, BNqMark presents the BN directly through serialized CPTs. This allows us to isolate exact BN inference more cleanly, study larger networks, and analyze sources of difficulty through structural properties such as treewidth and query total factor size. Other BN-related efforts, such as BIRD, use BNs to calibrate LLM confidence in downstream tasks (Feng et al., 2025). While adjacent, this differs from our objective: rather than building a Bayesian framework around an LLM, we use BNs to evaluate the LLM's own inference capabilities.

Tool-augmented reasoning and code generation. Recent work shows that LLM reasoning performance depends critically on whether computation is performed internally or externalized through tools or code. Approaches such as chain-of-thought prompting (Wei et al., 2022), self-consistency (Wang et al., 2023), and Tree-of-Thoughts (Yao et al., 2023a) improve reasoning by structuring intermediate steps, but they still rely on the model to execute computations itself. Tool-augmented methods instead allow models to externalize computation. Program-Aided Language Models (PAL) (Gao et al., 2023) demonstrate that generating executable code can substantially improve performance on mathematical tasks, while frameworks such as Toolformer (Schick et al., 2023) and ReAct (Yao et al., 2023b) enable models to interact with external tools during reasoning. These approaches motivate our separation between raw reasoning and code generation: the former tests whether models can carry out exact BN inference internally, while the latter tests whether they can externalize the same computation as executable code.

Mathematical reasoning benchmarks. BNqMark also fits within the broader effort to benchmark mathematical reasoning in LLMs. As reasoning-focused models have improved, widely used mathematical benchmarks have become increasingly saturated, motivating harder evaluation sets based on Olympiad-style problems, including OlympiadBench (He et al., 2024), Omni-MATH (Gao et al., 2024), and MathArena (Balunović et al., 2025). BNqMark differs from these benchmarks in both content and scoring target: each task is an exactly specified probabilistic computation rather than a free-form contest problem. Solv-



(a) **(b)**

Figure 1: Example of a categorical Bayesian network. (a) Graphical structure and CPTs. (b) Serialized CPT representation in the format presented to LLMs.

ing a query requires inferring the probabilistic structure encoded by the CPTs, identifying the required conditioning and marginalization operations, and carrying out the necessary sums and products. In BNqMark-20, this can require marginalizing over up to 18 hidden binary variables in the largest 1-target, 1-evidence setting, while exact algorithmic solutions remain available.

3. Probabilistic Inference in Bayesian Networks

3.1. Bayesian Networks

A BN is a pair (G, Θ), where G is a directed acyclic graph (DAG) over random variables $\mathbf{V}$ and Θ is a set of parameters specifying the conditional probability distributions (CPDs). The arcs of G encode conditional independencies between variables, allowing the joint distribution to factorize as $P(\mathbf{V}) = P(V_1, \ldots, V_n) = \prod_{i=1}^{n} P(V_i \mid \mathbf{Pa}_{V_i})$, where $n = |\mathbf{V}|$ is the number of variables and $\mathbf{Pa}_{V_i}$ denotes the parents of variable V_i in G. Figure 1 shows an example BN with four binary variables $\{V_0, V_1, V_2, V_3\}$ together with its CPTs in the serialized format presented to the models.

3.2. Exact Inference

Given a BN, we consider exact conditional inference queries of the form $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$, where $\mathbf{E} \subset \mathbf{V}$ denotes observed evidence variables with values $\mathbf{e}$ and $\mathbf{Q} \subseteq \mathbf{V} \setminus \mathbf{E}$ denotes

query variables with values $\mathbf{q}$. Using the factorized joint distribution, such queries can be written as

$$P(\mathbf{Q}=\mathbf{q} \mid \mathbf{E}=\mathbf{e}) = \frac{\sum_{\mathbf{z}} \prod_{i=1}^{n} P\big(V_i = v_i \mid \mathbf{Pa}_{V_i} = \mathbf{pa}_{V_i}\big)\,\big|_{(\mathbf{Q},\mathbf{E},\mathbf{Z})=(\mathbf{q},\mathbf{e},\mathbf{z})}}{\sum_{\mathbf{q}'} \sum_{\mathbf{z}} \prod_{i=1}^{n} P\big(V_i = v_i \mid \mathbf{Pa}_{V_i} = \mathbf{pa}_{V_i}\big)\,\big|_{(\mathbf{Q},\mathbf{E},\mathbf{Z})=(\mathbf{q}',\mathbf{e},\mathbf{z})}},$$

where $\mathbf{Z} = \mathbf{V} \setminus (\mathbf{Q} \cup \mathbf{E})$ are the unobserved variables, $\mathbf{q}'$ ranges over all assignments to the query variables, and $(v_i, \mathbf{pa}_{V_i})$ denotes the values induced by the full assignment in the corresponding term.

Efficient exact inference algorithms exploit this factorization rather than enumerating all assignments explicitly. A canonical example is variable elimination (VE), which successively sums out non-query variables by multiplying together the factors that contain them and then marginalizing the result (Shachter et al., 1990; Zhang and Poole, 1994).

Although exact inference in categorical BNs is NP-hard in general (Cooper, 1990), it becomes tractable in networks with bounded treewidth (Kwisthout et al., 2010). Treewidth measures how tree-like a graph is (Robertson and Seymour, 1986), and for binary variables VE runs in time exponential in the treewidth w, requiring intermediate factors of size up to $O(2^w)$. This makes treewidth a key graph-level parameter governing exact inference difficulty and the primary axis along which we vary graph-level complexity in BNqMark.

For a concrete inference task, however, complexity is not determined by the BN alone, and one can apply a number of query-specific simplifications. For example, variables that are independent of $\mathbf{Q}$ given $\mathbf{E}$ can be removed, and barren variables can be pruned. A variable is barren for a query if neither it nor any of its descendants belongs to $\mathbf{Q} \cup \mathbf{E}$; equivalently, it is not an ancestor of any target or evidence variable. Such variables can be removed by repeatedly pruning leaf variables outside $\mathbf{Q}$ and $\mathbf{E}$. Thus, two queries on the same BN can have substantially different effective inference complexity. We track the cost of a particular query with a number of measures, including the total factor size (Section 4.1).

4. Dataset

BNqMark is a dataset-generation framework for exact conditional inference in categorical BNs. The framework supports configurable graph structures, query families, and CPTs. In particular, it can vary parameters such as the number of variables, target treewidth, variable arity, and CPT concentration or determinism. It can also sample conditional probability queries under different target/evidence counts, distances, and informativeness constraints (e.g., discarding cases where evidence is d-separated from the target), while computing exact probabilities and query-complexity metadata.

Settings. In this study, we consider a particular instantiation of BNqMark, BNqMark-20, designed to balance structural diversity, exact solvability, and evaluation cost. The benchmark is test-only and intended for zero-shot evaluation, with no training or validation split. All variables in this dataset are binary, so difficulty is driven primarily by graph and query complexity rather than by variable cardinality. This restriction also keeps exact inference and large-scale model evaluation computationally manageable.

For BNqMark-20, network sizes range from 4 to 20 variables. For each even size n, we construct networks targeting treewidth values $tw \in \{2, 4, \ldots, \min(12, n-2)\}$, yielding

39 distinct (n, tw) configurations. For each configuration, we generate 2 network instances that share the same DAG but differ in their CPTs, resulting in 78 networks in total.

To vary the distributional properties of the CPTs, we sample each instance from a symmetric Dirichlet distribution with concentration parameter $\alpha \in \{1.0, 0.5\}$. The former produces relatively uniform probabilities, whereas the latter yields more skewed, near-deterministic distributions.

Graph generation. We generated graph structures by search-based sampling targeting a desired treewidth level, rather than sampling arbitrary random DAGs as in (Ide and Cozman, 2002; Scutari, 2010). Starting from a random tree (treewidth 1), we added and oriented edges while estimating treewidth with the minimum-degree heuristic for treewidth upper-bound estimation (Bodlaender and Koster, 2010). This search was repeated within a fixed budget, and we retained the candidate whose estimated treewidth was closest to the target.

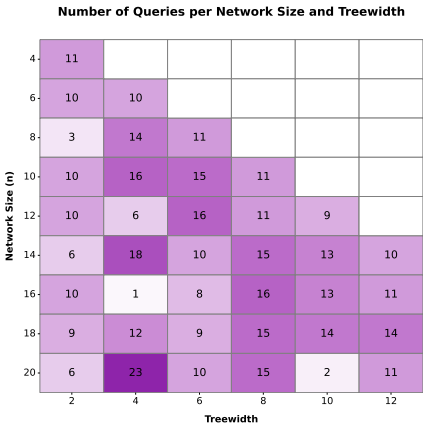
Query generation. From each of the 78 networks, we generate conditional probability queries of the form $P(\mathbf{Q} \mid \mathbf{E})$, where the query set $\mathbf{Q}$ contains 1 or 2 target variables and the evidence set $\mathbf{E}$ contains 1 or 2 observed variables. We consider all 4 combinations of target and evidence cardinalities and sample queries under distance thresholds, defined as the minimum number of undirected edges between $\mathbf{Q}$ and $\mathbf{E}$ of 1, 2, and 3. In this setting, larger distances generally require marginalizing over more intermediate variables.

For each network and each combination of query/evidence cardinality and distance stratum, we attempt to sample 1 valid query, retrying up to 50 times if the sampled variables do not satisfy the constraints. In addition, all queries must meet an informativeness threshold, $|P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}) - P(\mathbf{Q} = \mathbf{q})| \geq \delta$, with $\delta = 0.1$, which filters out cases where conditioning has little effect, so that simply outputting the marginal probability as the answer yields an absolute error of at least 0.1.

In total, this yields up to 12 query slots per network (4 cardinality combinations $\times$ 3 distance thresholds). Not all slots are filled within the sampling budget, so the final benchmark contains 434 queries out of 936 initial attempts. Ground-truth probabilities are computed using the `pgmpy` implementation of VE with a min-fill heuristic. Figure 2(a) shows the distribution of queries across treewidth and network size. Odd treewidth values arise from the approximate nature of the generation procedure, which targets but does not guarantee exact treewidth. Overall, the dataset spans a range of structural complexities: 40% of queries with low (2–4), 37% with medium (6–8), and 22% with high treewidth (10–12).

4.1. Query Complexity Quantification

For each generated query, BNqMark computes query-complexity metadata by simulating VE for that query. Prior to VE, we apply the query-specific simplifications described in Section 3.2: removing variables that are d-separated from every target variable given the evidence, and then iteratively pruning barren variables. The reduced BN is then analyzed using a VE order produced by `pgmpy`'s weighted min-fill heuristic, eliminating all non-target variables. We record the induced width of this order, defined as the maximum induced clique size minus one, and the query total factor size, defined as the sum of the intermediate factor



(a)

Model	Context	Ref
Claude Sonnet 4.6	1M	(Anthropic, 2026)
DeepSeek V3.2-Speciale	164K	(Liu et al., 2025)
Gemini 3.1 Pro	1M	(DeepMind, 2026)
GLM-5	1M	(Zeng et al., 2026)
GPT-5.4	1M	(Singh et al., 2026)
Grok 4.20	2M	(xAI, 2026)
Kimi K2.5	262K	(Bai et al., 2026)
MiniMax M2.7	200K	(MiniMax, 2026)
Qwen3-Max-Thinking	262K	(Yang et al., 2025)

(b)

Figure 2: (a) Number of queries by network size (y-axis) and treewidth (x-axis); each cell shows the query count for that combination. (b) Models evaluated in this work.

sizes generated during elimination, plus the final target-factor size. We also compute the total number of scalar additions and scalar multiplications required by this VE simulation for each query. Appendix A reports the distribution of these metrics.

5. Experiments

5.1. Evaluation Protocols

We evaluate models under 2 complementary protocols: *raw reasoning* and *code generation*. Both protocols use the same prompt structure: a system prompt (Appendix B.1) describing the CPT serialization format, followed by the serialized conditional probability tables, the query $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$, and a task prompt. The graph structure is never provided explicitly, so the model must infer the DAG from the CPT headers; Figure 1 illustrates the CPT format. No worked examples, chain-of-thought scaffolding, or BN terminology are included. The protocols differ only in the required output:

- **Raw reasoning.** The model must compute the conditional probability directly and return a final line of the form `Final Answer: P(...) = 0.xxx`. This tests whether it can carry out exact BN inference without external tools. The task prompt template is given in Appendix B.2.

- **Code generation.** The model must produce a self-contained Python program that computes and prints the answer. The code is executed in a sandboxed environment with access to standard libraries, `pgmpy`, and `pyAgrum`. This tests whether the model can externalize the inference computation as executable code, optionally using BN libraries. The task prompt template is given in Appendix B.3.

Table 1: Raw reasoning performance across LLMs on the BNqMark-20 benchmark.

Model	Queries	Answerability	Accuracy	MAE ± SD	RE mean (median)
Claude Sonnet 4.6	406	0.525 (8)	0.522 (7)	**0.001 ± 0.013 (1)**	**0.014 (0.001)**
DeepSeek V3.2-Speciale	345	0.501 (9)	0.461 (8)	0.013 ± 0.062 (4)	0.033 (0.001)
Gemini 3.1 Pro	402	0.908 (2)	0.789 (2)	0.012 ± 0.040 (3)	0.050 (0.001)
GLM-5	379	0.770 (5)	0.578 (6)	0.044 ± 0.109 (9)	0.161 (0.001)
GPT-5.4	**434**	0.825 (3)	**0.813 (1)**	0.002 ± 0.013 (2)	0.016 (0.001)
Grok 4.20	**434**	**0.963 (1)**	0.728 (3)	0.034 ± 0.083 (7)	0.117 (0.001)
Kimi K2.5	402	0.774 (4)	0.597 (5)	0.041 ± 0.097 (8)	0.140 (0.001)
MiniMax M2.7	379	0.583 (7)	0.451 (9)	0.033 ± 0.097 (6)	0.180 (0.001)
Qwen3-Max-Thinking	402	0.721 (6)	0.602 (4)	0.022 ± 0.072 (5)	0.069 (0.001)

5.2. Models and Metrics

We evaluate 9 reasoning-focused LLMs spanning multiple providers, see Figure 2(b). All models are accessed through OpenRouter (OpenRouter, 2026) with temperature 1.0 and maximum available reasoning effort (`xhigh`); Appendix C lists the exact API model strings and provider preferences used for each model. Each model is evaluated on all 434 queries under both protocols described in Section 5.1. We report 4 metrics:

Answerability. The fraction of queries for which the model produces a valid, extractable numerical answer. In the raw-reasoning protocol, this requires a correctly formatted final-answer line. In the code-generation protocol, it requires code that executes without error and prints a parseable probability.

Accuracy. The fraction of evaluated queries for which $|p_{\text{model}} - p_{\text{true}}| \leq \tau$, where p_{model} is the extracted model probability, p_{true} is the exact ground truth, and $\tau = 0.01$. Queries without a valid extracted answer are counted as incorrect.

Mean absolute error (MAE). The average of $|p_{\text{model}} - p_{\text{true}}|$ over answerable queries only. We also report the standard deviation (SD) of the absolute error.

Relative error (RE). For each answerable query, we compute the RE $= |p_{\text{model}} - p_{\text{true}}|/p_{\text{true}}$, and report its mean and median.

We compute all metrics only on queries that fit within each model's context length limit. The metrics capture different aspects of performance: answerability measures whether the model produces a usable output at all; accuracy captures end-to-end success by treating non-answerable cases as failures; and MAE and RE measure numerical precision, conditional on producing a valid answer.

6. Results

Several frontier LLMs can answer a substantial fraction of exact BN inference queries without worked examples, especially when using code generation. For example, GPT-5.4 correctly answered 81.3% of the queries without access to tools (Table 1) and 87.6% when allowed to generate code (Table 2). Although performance declined as graph-level and query-level complexity increased (Appendices D, E, and F), several models still answered fairly complex queries. In the code-generation protocol, GPT-5.4 correctly answered 9/17 queries that required between 5,000 and 10,000 scalar arithmetic operations in our VE simulation, as well as 1 query that required 68,687 operations. In raw reasoning, Gemini 3.1 Pro and GLM-5 correctly answered a query that required 13,083 operations.

Table 2: Code generation performance across LLMs on the BNqMark-20 benchmark.

Model	Queries	Answerability	Accuracy	MAE ± SD	RE mean (median)
Claude Sonnet 4.6	403	0.809 (6)	0.809 (4)	**0.000 ± 0.000 (1)**	**0.005 (0.001)**
DeepSeek V3.2-Speciale	310	0.619 (8)	0.590 (8)	0.014 ± 0.082 (7)	0.050 (0.001)
Gemini 3.1 Pro	403	0.866 (2)	0.854 (2)	0.004 ± 0.047 (4)	0.039 (0.001)
GLM-5	379	0.649 (7)	0.625 (7)	0.004 ± 0.029 (3)	0.020 (0.001)
GPT-5.4	**434**	**0.882 (1)**	**0.876 (1)**	0.001 ± 0.002 (2)	0.006 (0.001)
Grok 4.20	**434**	0.846 (3)	0.820 (3)	0.005 ± 0.032 (5)	0.025 (0.001)
Kimi K2.5	402	0.828 (4)	0.749 (6)	0.040 ± 0.356 (9)	0.102 (0.001)
MiniMax M2.7	379	0.612 (9)	0.562 (9)	0.013 ± 0.059 (6)	0.042 (0.001)
Qwen3-Max-Thinking	402	0.821 (5)	0.754 (5)	0.016 ± 0.068 (8)	0.060 (0.001)

While we considered an answer to be correct when within $\tau = 0.01$ of the ground truth, accuracies would have barely dropped with $\tau = 0.001$ or $\tau = 0.0005$, especially in code generation (e.g., from 0.876 to 0.862 for GPT-5.4), as also suggested by the low MAE and RE (e.g., 0.001 and 0.006 for GPT-5.4 in code generation).

Overall, 3 main patterns emerge: a) Externalizing inference as executable code improves accuracy for all models, although the mechanisms by which models benefit vary and answerability does not always improve; b) Both treewidth and query total factor size predict answerability and accuracy after controlling for prompt length, network size, and model identity, with total factor size most strongly related to accuracy (Section 6.3); and c) The two protocols reveal different relationships between answerability and accuracy: in raw reasoning, models more often produce formatted but numerically incorrect answers, whereas in code generation valid outputs are more likely to be accurate once produced.

6.1. Raw Reasoning

The raw-reasoning protocol shows that several frontier LLMs are already capable of strong exact-inference performance even without external tools. Table 1 shows that GPT-5.4 achieves the highest aggregate accuracy (0.813), with Gemini 3.1 Pro close behind (0.789), although these comparisons must be read alongside the "Queries" column because some models are evaluated on fewer queries due to context length constraints. The gap between answerability and accuracy also reveals distinct failure modes. Grok 4.20 attains the highest answerability (0.963) but substantially lower accuracy (0.728), indicating that it usually returns a well-formed answer yet often makes numerical mistakes. Claude Sonnet 4.6 exhibits the opposite profile: it answers only about half of the queries, but when it does answer, it is almost always correct, yielding nearly identical answerability and accuracy and the lowest MAE on answerable queries.

Appendix D presents a detailed view of raw-reasoning accuracy across treewidth and query total factor size. In the raw-reasoning accuracy heatmap (Figure 5), several models remain strong in lower-complexity cells, whereas performance deteriorates as treewidth and total factor size increase. Similar patterns appear in the answerability and MAE heatmaps (Figures 7 and 9). However, the regression analysis in Section 6.3 makes the distinction clearer: both treewidth and query total factor size affect answerability and accuracy, while total factor size is especially tied to numerical correctness on computationally demanding queries.

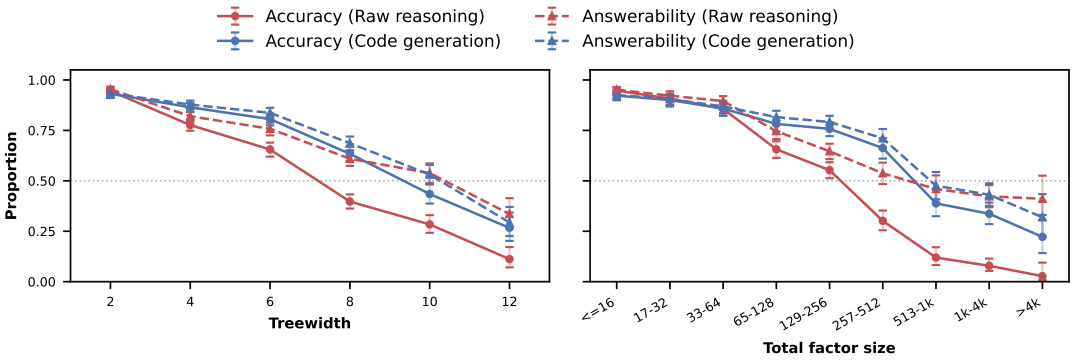


Figure 3: Pooled accuracy (solid) and answerability (dashed) versus treewidth (left) and total factor size (right), for raw reasoning (red) and code generation (blue), with Wilson 95% confidence intervals.

6.2. Code Generation

Table 2 shows that code generation improves end-to-end performance. Accuracy improves for every model, although answerability does not universally improve, suggesting that executable externalization helps with inference while also introducing new opportunities for formatting and runtime failures. GPT-5.4 again performs best overall, reaching 0.876 accuracy and 0.882 answerability, with Gemini 3.1 Pro (0.854) and Grok 4.20 (0.820) close behind. More broadly, answerability and accuracy become much closer for the strongest LLMs, and MAE drops to near zero, indicating that once a model successfully produces runnable code, the resulting answer is usually correct. Claude Sonnet 4.6 illustrates this pattern especially clearly: its answerability and accuracy are both 0.809, with effectively zero MAE, implying that its remaining errors arise primarily before successful execution rather than from incorrect inference once the program runs.

Appendix G.2 shows that this gain does not arise from a single strategy. Most models use both `pgmpy` and plain or general-purpose Python, with roughly similar conditional accuracy. GPT-5.4's answerable generations all fall into the no-BN-library category and nevertheless reach 99.22% accuracy. The code-generation advantage therefore reflects a broader ability to externalize numeric computation into executable programs, not just the correct formulation of a problem for a dedicated BN solver. At the same time, Figure 3 shows that code generation mitigates, but does not remove, the effect of inference complexity: in the per-model accuracy heatmap (Figure 6), lower-treewidth and lower-factor-size examples are solved almost perfectly by the best models, but accuracy still deteriorates as either complexity axis increases and eventually gives way to context-length limits. Executable externalization thus substantially reduces inference errors, while the dependence on BN structure and query-specific inference burden remains.

6.3. Prompt Length vs. Graph-Level and Query-Level Complexity

Figure 3 shows performance declines along both treewidth and total factor size. However, higher-treewidth networks produce longer prompts, and queries sharing the same treewidth can differ in their induced factor sizes after conditioning and pruning. We separate these effects by fitting pooled logistic regressions for answerability and accuracy on supported examples, using treewidth, total factor size, input tokens, and network size as predictors, with model-specific intercepts (Table 3).

Across both protocols, treewidth and total factor size were negatively associated with answerability and accuracy. Treewidth mattered more for answerability, whereas total factor size mattered more for accuracy: in raw reasoning its coefficient is -5.389 versus -0.908 for treewidth, and in code generation the two are comparable (-0.984 vs. -1.009). This makes sense, since answerability reflects context and formatting failures, while accuracy depends on correctly carrying out the numerical inference, which is harder in raw reasoning.

Input length also had a negative but smaller effect than the main complexity terms, and network size contributed little once the other predictors were included. These results suggest that the observed decline is not merely a prompt-length artifact or a function of node count alone, but reflects both graph-level and query-level aspects of exact-inference complexity. Per-model regressions (Appendix H.1) show heterogeneity in which difficulty signal dominates: total factor size affects code-generation accuracy for several models, whereas GPT-5.4 is more clearly affected by input length. The pooled result, however, is that query complexity matters in addition to treewidth.

6.4. Preliminary Non-Black-Box and Failure Analysis

The visible raw-reasoning trace consists of generated summaries rather than direct records of the computations and as such is not necessarily faithful to the reasoning process. Non-black-box analysis, therefore, only makes sense for the code-generation protocol. Appendix G.1 shows failure statistics, while Appendix G.3 shows 2 models' solutions to a query. In that example, GPT-5.4 generated a plain-Python solution that carefully exploited the network structure instead of blindly enumerating all assignments: it pruned 2 barren nodes and enumerated only the necessary assignments to the 4 remaining unobserved variables. It used 32 additions plus 96 multiplications, close to the 31 additions plus 92 multiplications required by the VE after conditioning and pruning. The main failure modes were missing executable code, output-contract violations, and malformed BN or CPT representations; only 21 failures were due to syntax or indentation errors (Appendix G.1).

7. Conclusion

Summary. We studied the extent to which current LLMs can perform exact probabilistic inference when the probabilistic model is fully specified. Using BNs as a controlled setting, we introduced BNqMark to enable a systematic evaluation of exact conditional inference while varying graph-level and query complexity. Our results show that frontier reasoning LLMs can solve many exact inference queries on small and medium BNs, especially when they externalize computation as code, while raw (tool-free) reasoning is reliable mainly on simpler queries. Performance remains strongly bounded by the complexity of inference.

Table 3: Pooled logistic regressions of answerability and accuracy on treewidth, total factor size, input tokens, and network size (all standardized), with model-specific intercepts. AUC and McFadden pseudo-R^2 summarize goodness of fit. $^{***}p < 0.001$, $^{**}p < 0.01$.

Protocol	**Outcome**	**N**	β_{tw}	β_{tok}	β_n	β_{fs}	**AUC**	**McFadden R^2**
Raw reasoning	Answerability	3583	-0.734^{***}	-0.676^{***}	-0.242^{***}	-0.464^{***}	0.783	0.309
Raw reasoning	Accuracy	3583	-0.908^{***}	-0.274^{***}	-0.021	-5.389^{***}	0.851	0.413
Code generation	Answerability	3546	-0.943^{***}	-0.192^{***}	0.046	-0.575^{***}	0.799	0.250
Code generation	Accuracy	3546	-1.009^{***}	-0.249^{***}	0.164^{**}	-0.984^{***}	0.820	0.291

Limitations. BNqMark-20 is intentionally designed as a controlled benchmark, and this focus imposes several limitations. First, the current release is restricted to categorical BNs with binary variables, network sizes up to 20 nodes, and exact conditional inference queries. These choices let us isolate graph-level and query complexity cleanly and compute exact ground truth at scale, but they limit how directly the results generalize to larger networks, variables with higher cardinality, or approximate inference settings. Second, our evaluation is single-turn and zero-shot: models receive a fully specified network and must either compute the answer directly or generate code in one pass. This setup is useful for measuring internal computation and structured problem translation in isolation, but it does not capture agentive, multi-step workflows. Finally, because we asked the LLMs to report probabilities to only three decimal places, our study does not assess higher-precision numerical performance. It also provides an incomplete picture of relative error: with at most two evidence variables, the benchmark contains few low-probability queries, only 6 of the 434 ground-truth probabilities being below 1%. Nevertheless, we expect the code-generation results to remain largely robust under higher-precision evaluation and on queries with lower ground-truth probabilities.

Future work. An immediate next step is to extend the case study in Appendix G.3 into a systematic semantic audit of generated programs. It would also be valuable to extend the benchmark to larger networks, higher-cardinality variables, and continuous or hybrid BNs, and to move beyond single-turn prompting by wrapping raw-reasoning attempts in a verification-and-refinement pipeline (Huang and Yang, 2025) that checks and iteratively corrects the answer.

Data and Code Availability

Code and data are available at https://github.com/ferjorosa/bnqmark and https://huggingface.co/datasets/ferjorosa/bnqmark-20.

Acknowledgements

Bojan Mihaljević acknowledges funding from Comunidad de Madrid through the grant agreement to foster and promote research and technology transfer at the Universidad Politécnica de Madrid, under Línea A, Doctores Emergentes.

References

A. Ankan and J. Textor. pgmpy: A Python toolkit for Bayesian networks. *Journal of Machine Learning Research*, 25(265):1–8, 2024.

Anthropic. Claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026.

T. Bai, Y. Bai, Y. Bao, S. H. Cai, Y. Cao, et al. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

M. Balunović, J. Dekoninck, I. Petrov, N. Jovanović, and M. Vechev. MathArena: Evaluating LLMs on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025.

H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations I. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

G. Cooper. Probabilistic inference using belief networks is NP-hard. *Artificial Intelligence*, 42:393–405, 1990.

DeepMind. Gemini 3.1 pro. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/, 2026.

G. Ducamp, C. Gonzales, and P.-H. Wuillemin. aGrum/pyAgrum: a toolbox to build models and algorithms for probabilistic graphical models in Python. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models (PGM)*, pages 609–612, 2020.

Y. Feng, B. Zhou, W. Lin, and D. Roth. Bird: A trustworthy Bayesian inference framework for large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

B. Gao, F. Song, Z. Yang, Z. Cai, Y. Miao, Q. Dong, L. Li, C. Ma, L. Chen, R. Xu, Z. Tang, B. Wang, D. Zan, S. Quan, G. Zhang, L. Sha, Y. Zhang, X. Ren, T. Liu, and B. Chang. Omni-MATH: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024.

L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 10764–10799, 2023.

C. He, R. Luo, Y. Bai, S. Hu, Z. L. Thai, J. Shen, J. Hu, X. Han, Y. Huang, Y. Zhang, J. Liu, L. Qi, Z. Liu, and M. Sun. OlympiadBench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*, 2024.

Y. Huang and L. F. Yang. Winning gold at IMO 2025 with a model-agnostic verification-and-refinement pipeline. *arXiv preprint arXiv:2507.15855*, 2025.

J. S. Ide and F. G. Cozman. Random generation of Bayesian networks. In *Advances in Artificial Intelligence: 16th Brazilian Symposium on Artificial Intelligence (SBIA)*, volume 2507 of *Lecture Notes in Computer Science*, pages 366–376. Springer, 2002.

D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

J. Kwisthout, H. L. Bodlaender, and L. C. van der Gaag. The necessity of bounded treewidth for efficient inference in Bayesian networks. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 237–242, 2010.

B. Y. Lin, R. Le Bras, K. Richardson, A. Sabharwal, R. Poovendran, P. Clark, and Y. Choi. ZebraLogic: On the scaling limits of LLMs for logical reasoning. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 37889–37905, 2025.

A. Liu, A. Mei, B. Lin, B. Xue, B. Wang, B. Xu, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

MiniMax. MiniMax M2.7. https://www.minimax.io/models/text/m27, 2026.

A. Nafar, K. B. Venable, and P. Kordjamshidi. Reasoning over uncertain text by generative large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24911–24920, 2025.

OpenRouter. OpenRouter. https://openrouter.ai/, 2026.

A. Paruchuri, J. Garrison, S. Liao, J. B. Hernandez, J. Sunshine, T. Althoff, X. Liu, and D. McDuff. What are the odds? language models are capable of probabilistic reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 11712–11733, 2024.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

L. Qiu, F. Sha, K. Allen, Y. Kim, T. Linzen, and S. van Steenkiste. Bayesian teaching enables probabilistic reasoning in large language models. *Nature Communications*, 17: 1238, 2026.

N. Robertson and P. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

T. Schick, J. Dwivedi-Yu, R. Dessí, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 68539–68551, 2023.

M. Scutari. Learning Bayesian networks with the `bnlearn` R package. *Journal of Statistical Software*, 35(3):1–22, 2010.

R. Shachter, B. D. D'Ambrosio, and B. del Favero. Symbolic probabilistic inference in belief networks. In *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 126–131, 1990.

A. Singh, A. Fry, A. Perelman, A. Tart, A. Ganesh, A. El-Kishky, et al. OpenAI GPT-5 system card. *arXiv preprint arXiv:2601.03267*, 2026.

X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 24824–24837, 2022.

xAI. Grok 4.20. https://grok.com/release-notes/mar-03-2026, 2026.

A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 11809–11822, 2023a.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.

A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, et al. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

N. L. Zhang and D. Poole. A simple approach to Bayesian network computations. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.

Appendix A. Query-Complexity Statistics

This appendix reports query-complexity statistics for the 434 BNqMark-20 queries in our dataset. The statistics are computed using the query-specific pruning and weighted-min-fill VE simulation described in Section 3.2. Scalar summations are reported as scalar additions, and total scalar operations are the sum of scalar additions and scalar multiplications.

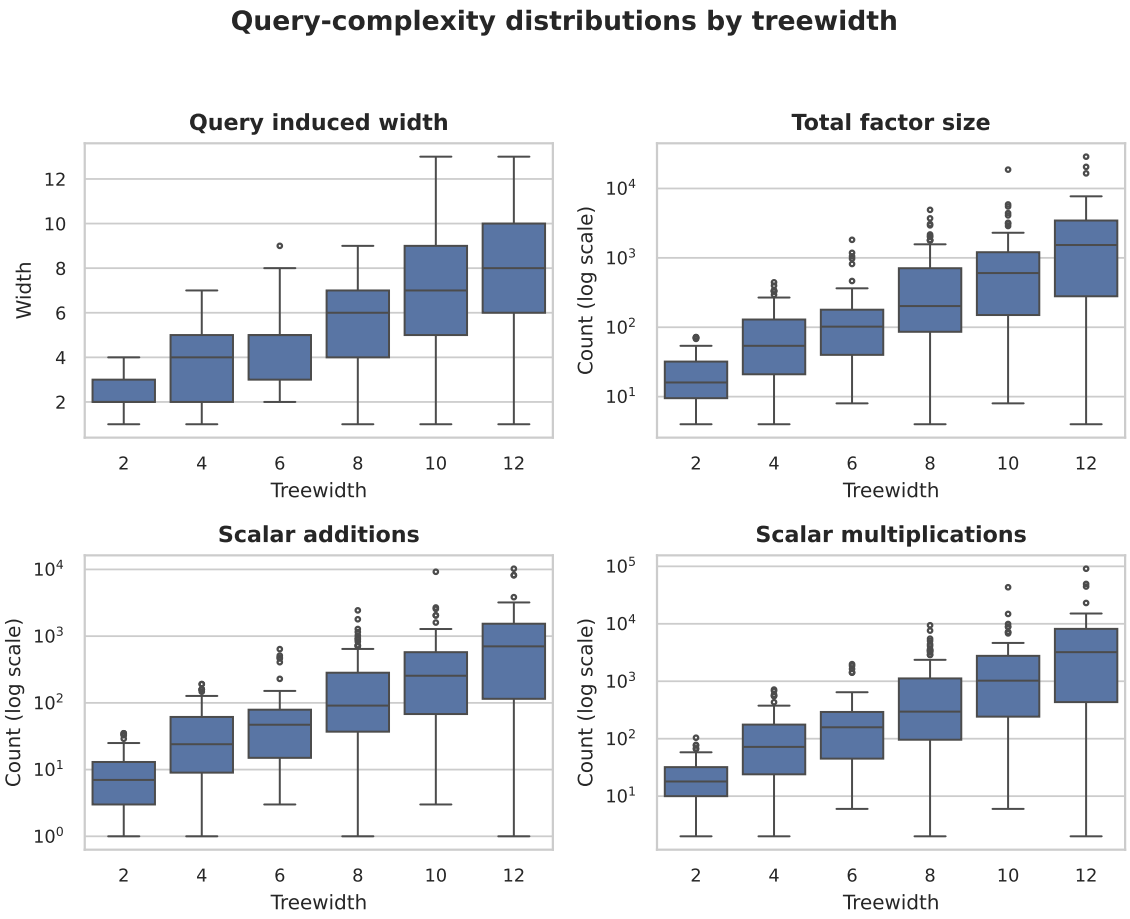


Figure 4: Distribution of query-complexity metrics by treewidth. Query total factor size is the sum of intermediate factor sizes generated by VE plus the final target-factor size. Scalar additions and scalar multiplications count dense tabular VE operations. Factor-size and operation-count panels use logarithmic y-axes because the distributions are heavy-tailed.

Appendix B. Prompt Templates - Probabilistic Inference

Both evaluation protocols share the same system prompt, which establishes the CPT format conventions. They differ only in the task prompt, which specifies whether the model should return a direct probability answer (raw reasoning) or executable Python code (code generation).

B.1. System Prompt

The system prompt introduces the model's role and provides a format guide for reading conditional probability tables. No BN terminology is used, and no worked examples of probabilistic reasoning are provided. The model is expected to infer the network structure and probabilistic meaning of the CPTs from the format description alone.

The examples in the format guide use semantically meaningful labels (`yes/no`, `A/B/C`) for clarity, while actual benchmark queries use neutral labels (`s0/s1`, `V0/V1`, ...) to avoid semantic priming.

```
You are an assistant that helps with probability reasoning calculations. You will be given
some conditional probability tables (CPTs) and a query.

## CPT Format Guide

### 1. Marginal CPTs (nodes with no parents)
‘‘‘
+----------------+------------+
| Node(Value)    | Probability|
+----------------+------------+
| A(yes)         | 0.3        |
+----------------+------------+
| A(no)          | 0.7        |
+----------------+------------+
‘‘‘
Each row means **P(Node=Value) = Probability**.

### 2. Conditional CPTs (nodes with one or more parents)
- The **first rows** list all parent nodes
- Each **column** represents one combination of parent values
- **Child rows** give P(child=value | parent assignment) for that column
- Probabilities in each column **sum to 1**

**Example:**
‘‘‘
+----------+----------+----------+----------+----------+
| A        | A(yes)   | A(yes)   | A(no)    | A(no)    |
+----------+----------+----------+----------+----------+
| B        | B(yes)   | B(no)    | B(yes)   | B(no)    |
+----------+----------+----------+----------+----------+
| C(yes)   | 0.8      | 0.6      | 0.4      | 0.1      |
+----------+----------+----------+----------+----------+
| C(no)    | 0.2      | 0.4      | 0.6      | 0.9      |
+----------+----------+----------+----------+----------+
‘‘‘
**Interpretation:** Each column represents one parent assignment. Child rows give
probabilities given that parent assignment.
```

B.2. Task Prompt: Raw Reasoning Protocol

Under the raw-reasoning protocol, the model is asked to compute the exact probability directly, without writing code. The `{cpts}` and `{query}` placeholders are filled at runtime with the serialized CPT tables and the specific query.

```
## Task

Below you will find conditional probability tables and a query. Compute the probability for
the following query.

**Conditional Probability Tables (CPTs):**

{cpts}

## Query

{query}

## Output Format

End your response with this line in exactly this format:

Final Answer: P(<target_node>=<value> | <evidence>) = <numeric_probability>

Where <numeric_probability> is the final probability rounded to 3 decimal places.
The final answer line must contain only the probability value | no trailing text, spaces,
or punctuation.
```

B.3. Task Prompt: Code Generation Protocol

Under the code-generation protocol, the prompt structure is identical, but the model is asked to produce executable Python code. The code is extracted from `<code></code>` tags and executed in a sandboxed environment.

```
## Task

Below you will find conditional probability tables and a query. Write Python code to compute
the probability for the following query.

**Conditional Probability Tables (CPTs):**

{cpts}

## Query

{query}

## Output Format

Provide your Python code between <code></code> tags:

<code>
# Your Python code here
# Use print() to output the numeric probability (a decimal between 0 and 1)
</code>

Output format requirements:
- Place your code between <code></code> tags
- The code must print only the numeric probability when executed
- You have access to standard Python libraries, pgmpy, and pyAgrum
- Print the final probability rounded to 3 decimal places
```

Appendix C. Exact Model Identifiers and Inference Configuration

All models were accessed through the OpenRouter API (OpenRouter, 2026) using the exact API model strings listed in Table 4. For each call, we set OpenRouter's reasoning effort to `xhigh` with a detailed reasoning summary, and we specified a provider preference (when available) so that the model is served by the router through the provider named in the rightmost column of Table 4.

Table 4: Exact OpenRouter API model strings used for each evaluated model, together with the provider preference used for routing (when one was set). Models without a provider preference were routed through OpenRouter's default selection.

Model	**OpenRouter API string**	**Provider preference**
Claude Sonnet 4.6	`anthropic/claude-sonnet-4.6`	`anthropic`
DeepSeek V3.2-Speciale	`deepseek/deepseek-v3.2-speciale`	`atlas-cloud/fp8`
Gemini 3.1 Pro	`google/gemini-3.1-pro-preview`	`google-ai-studio`
GLM-5	`z-ai/glm-5`	`z-ai`
GPT-5.4	`openai/gpt-5.4`	—
Grok 4.20	`x-ai/grok-4.20`	`xai`
Kimi K2.5	`moonshotai/kimi-k2.5`	`atlas-cloud/int4`
MiniMax M2.7	`minimax/minimax-m2.7`	`minimax/highspeed`
Qwen3-Max-Thinking	`qwen/qwen3-max-thinking`	`alibaba`

Appendix D. Accuracy Heatmaps

In all heatmap appendices, black cells labeled "N/A" indicate that the prompt exceeded the model's context length limit.

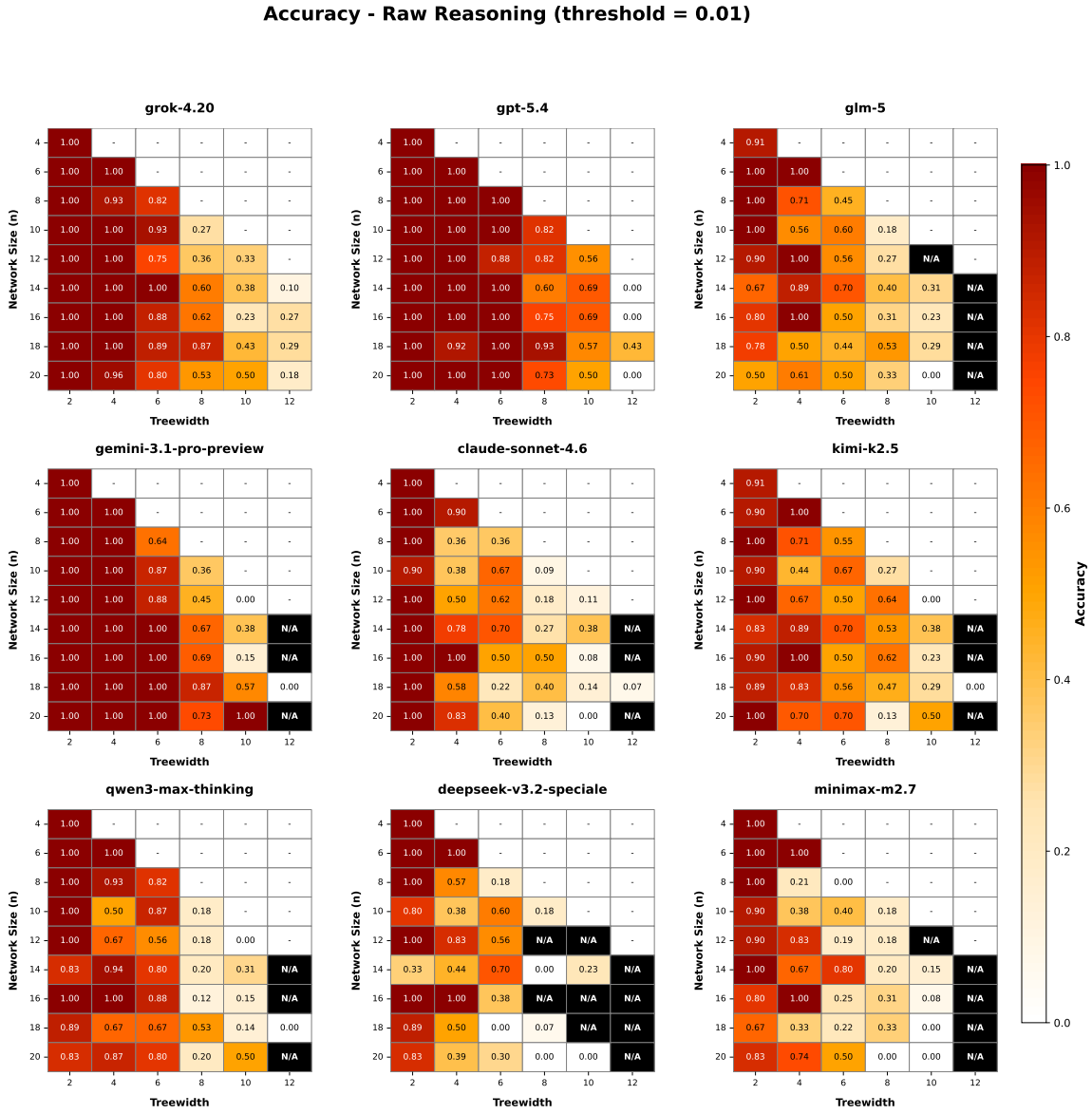


Figure 5: Accuracy ($\tau = 0.01$) by total factor size bin (y-axis) and treewidth (x-axis) in the raw-reasoning protocol. These are the strongest graph-level and query-level complexity axes for raw-reasoning accuracy in the pooled regression analysis. Subplots correspond to models; darker red indicates higher accuracy. White "–" cells denote missing query combinations, and black "N/A" cells denote context-limited settings.

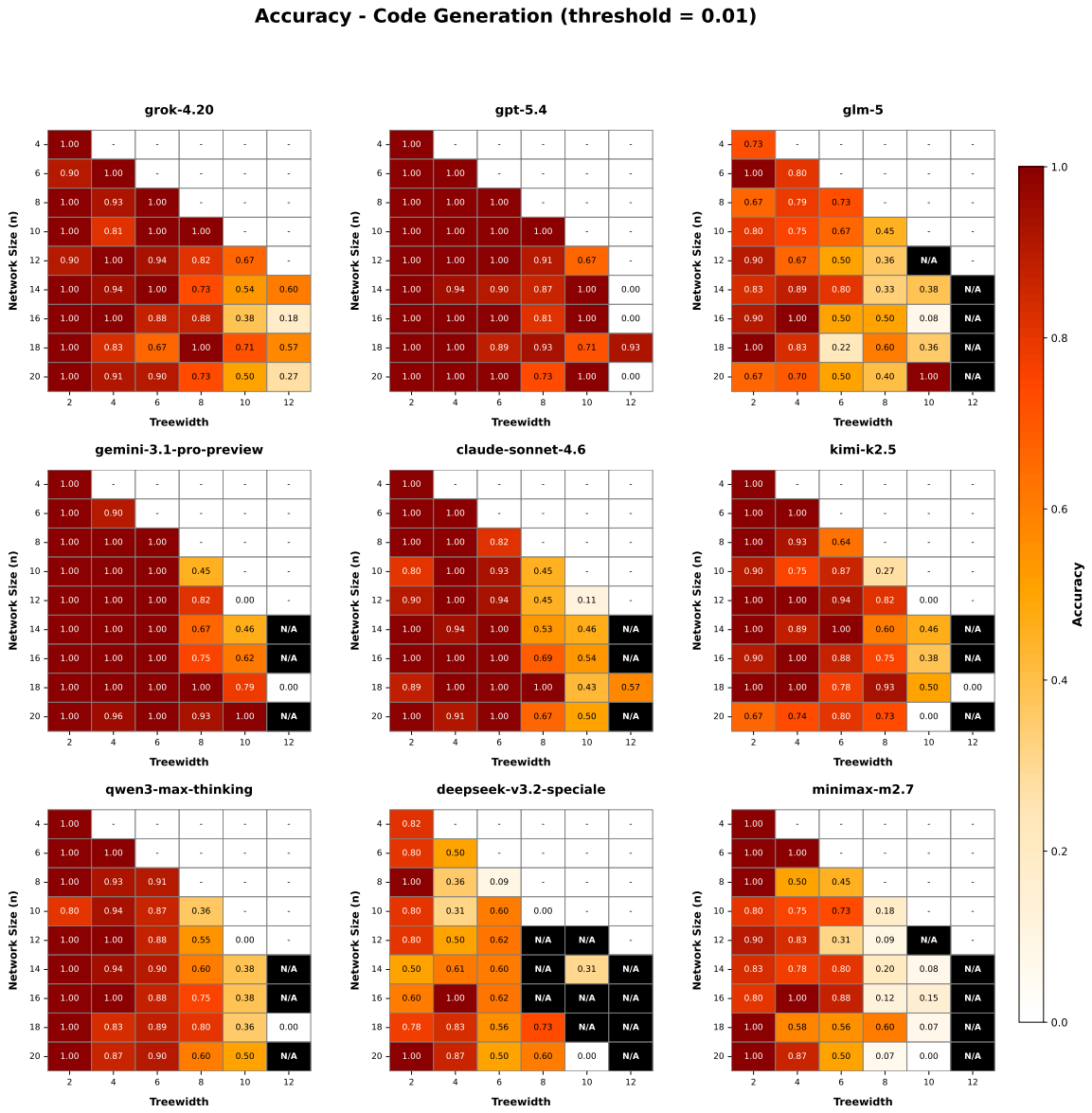


Figure 6: Accuracy ($\tau = 0.01$) by total factor size bin (y-axis) and treewidth (x-axis) in the code-generation protocol. These are the strongest graph-level and query-level complexity axes for code-generation accuracy in the pooled regression analysis. Subplots correspond to models; darker red indicates higher accuracy. White "–" cells denote missing query combinations, and black "N/A" cells denote context-limited settings.

Appendix E. Answerability Heatmaps

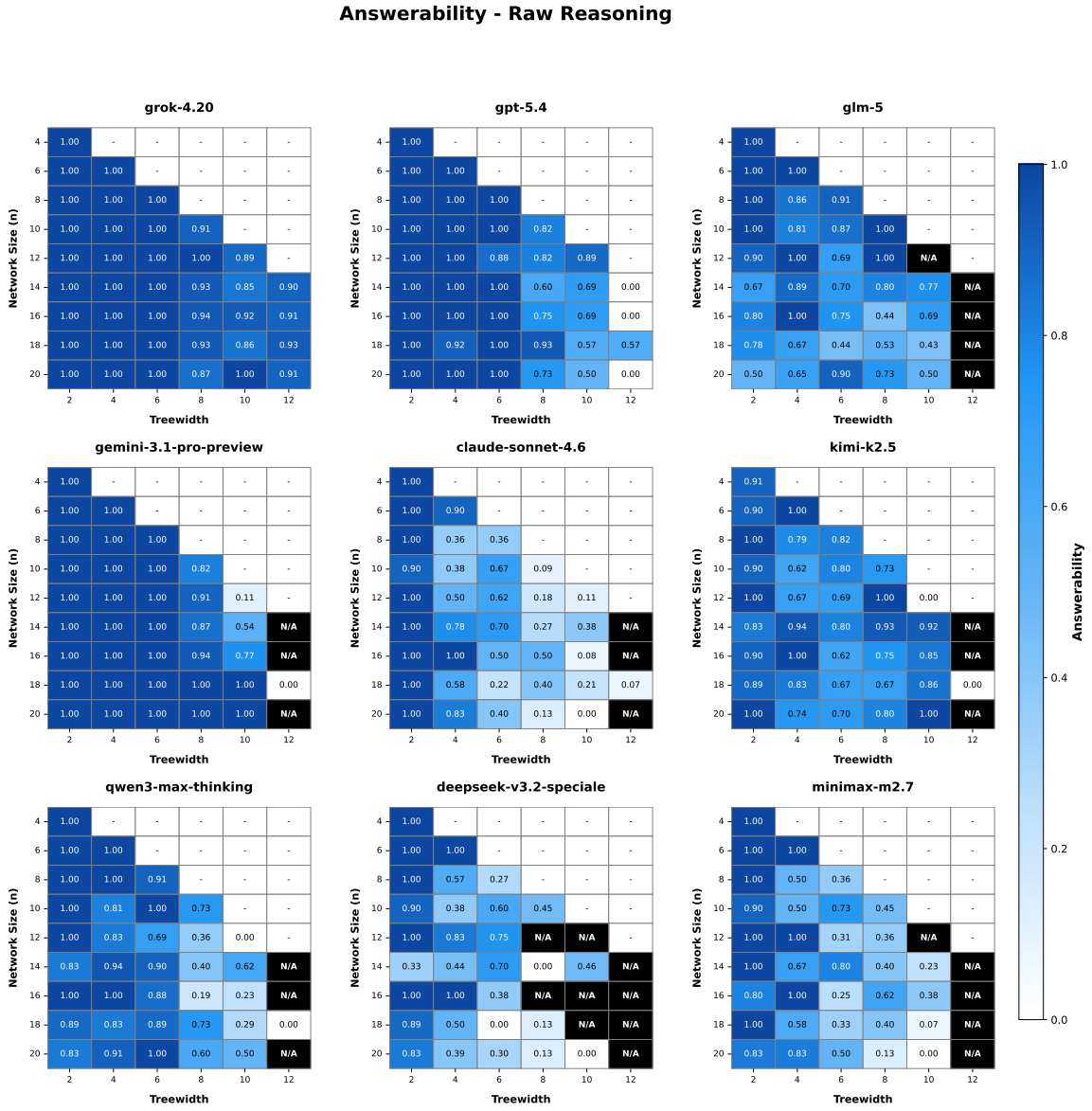


Figure 7: Answerability by total factor size bin (y-axis) and treewidth (x-axis) in the raw-reasoning protocol. These are the graph-level and query-level complexity predictors in the corresponding pooled answerability regression. Subplots correspond to models; darker blue indicates higher answerability. White "–" cells denote missing query combinations, and black "N/A" cells denote context-limited settings.

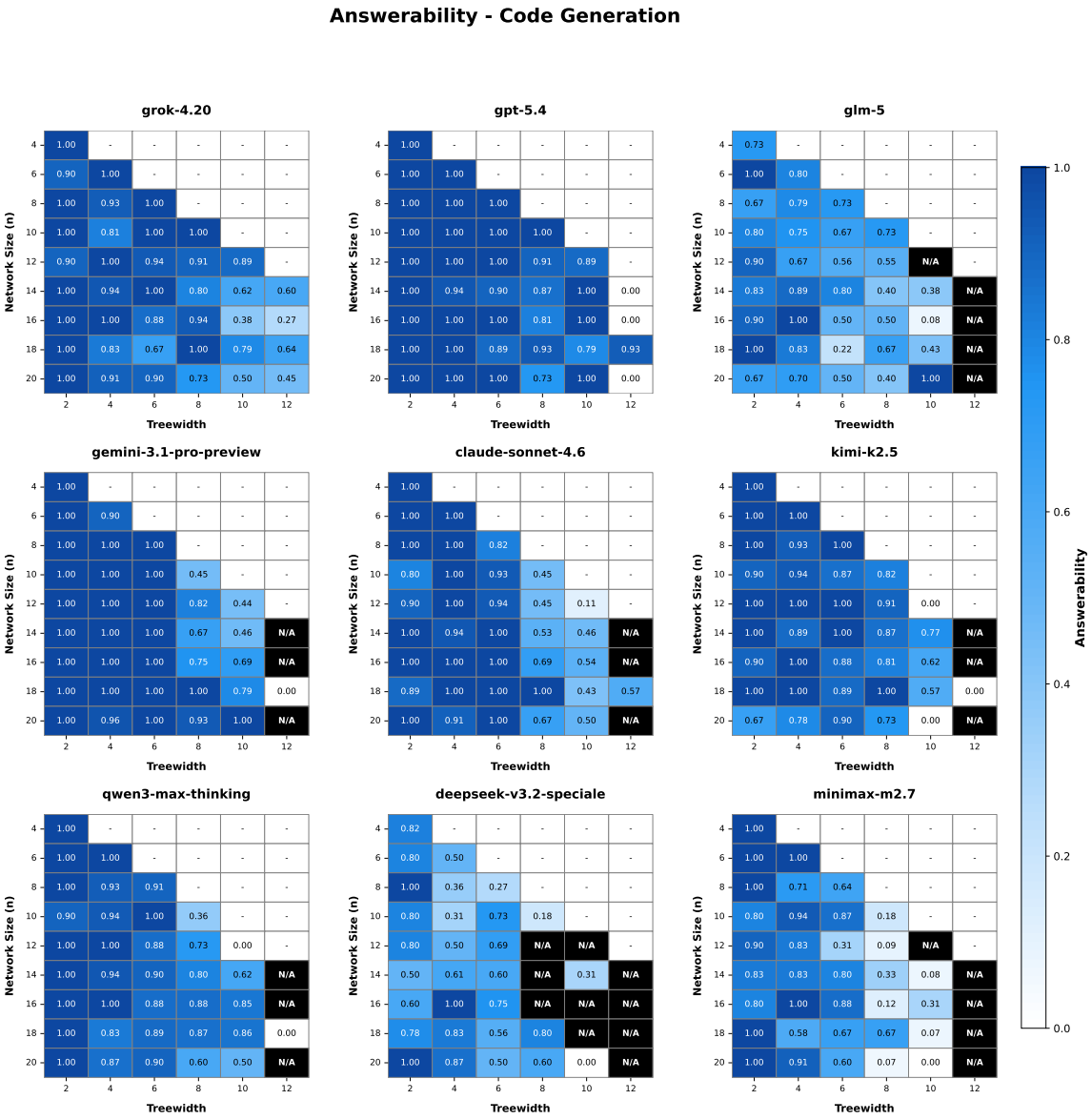


Figure 8: Answerability by total factor size bin (y-axis) and treewidth (x-axis) in the code-generation protocol. These are the graph-level and query-level complexity predictors in the corresponding pooled answerability regression. Subplots correspond to models; darker blue indicates higher answerability. White "–" cells denote missing query combinations, and black "N/A" cells denote context-limited settings.

Appendix F. Mean Absolute Error (MAE) Heatmaps

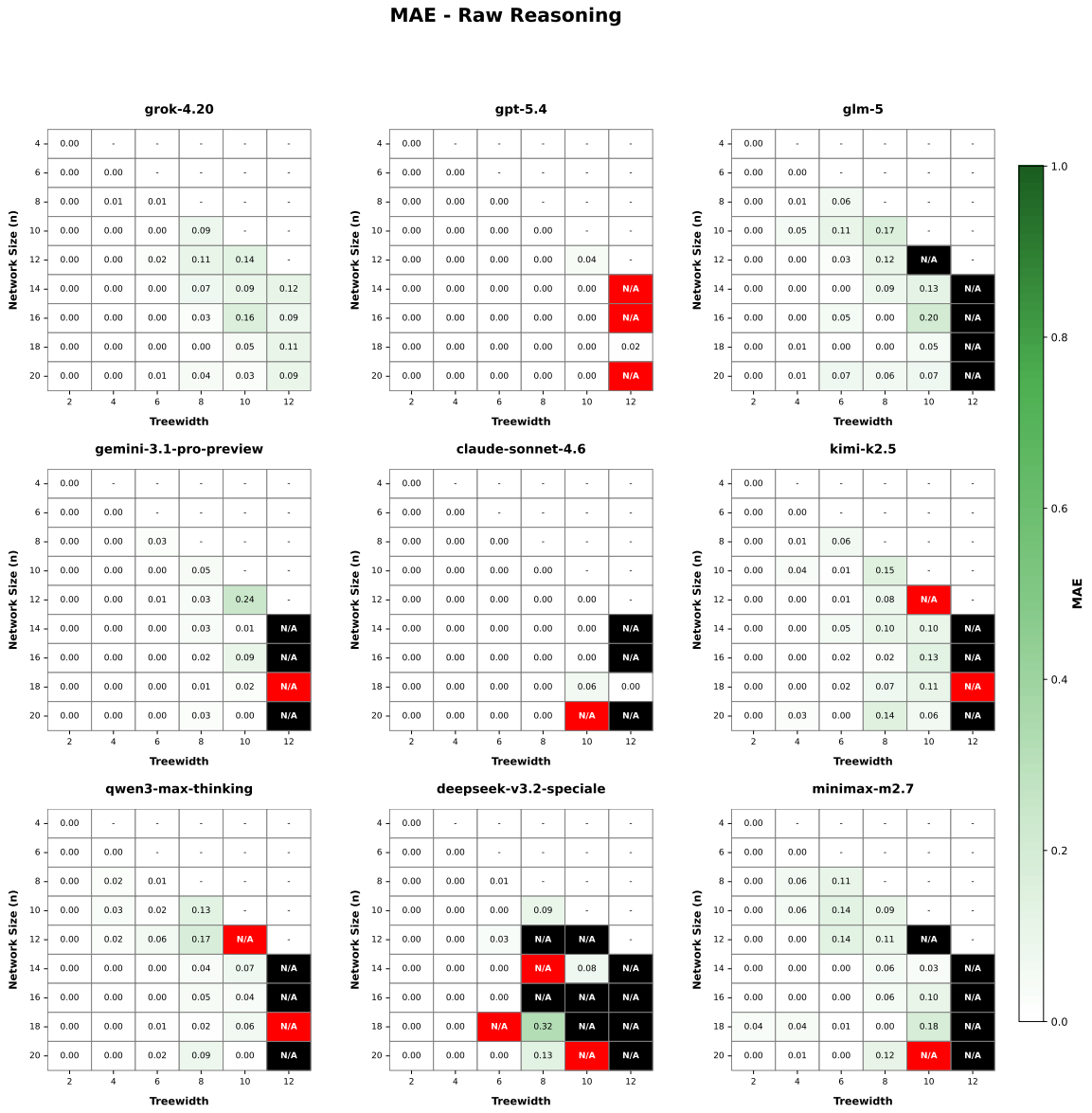


Figure 9: MAE by total factor size bin (y-axis) and treewidth (x-axis) in the raw-reasoning protocol, using the same graph-level and query-level complexity axes as the raw-reasoning accuracy analysis. Subplots correspond to models; darker green indicates higher error. White "–" cells denote missing query combinations, black "N/A" cells denote context-limited settings, and red "N/A" cells denote settings with no answerable queries.

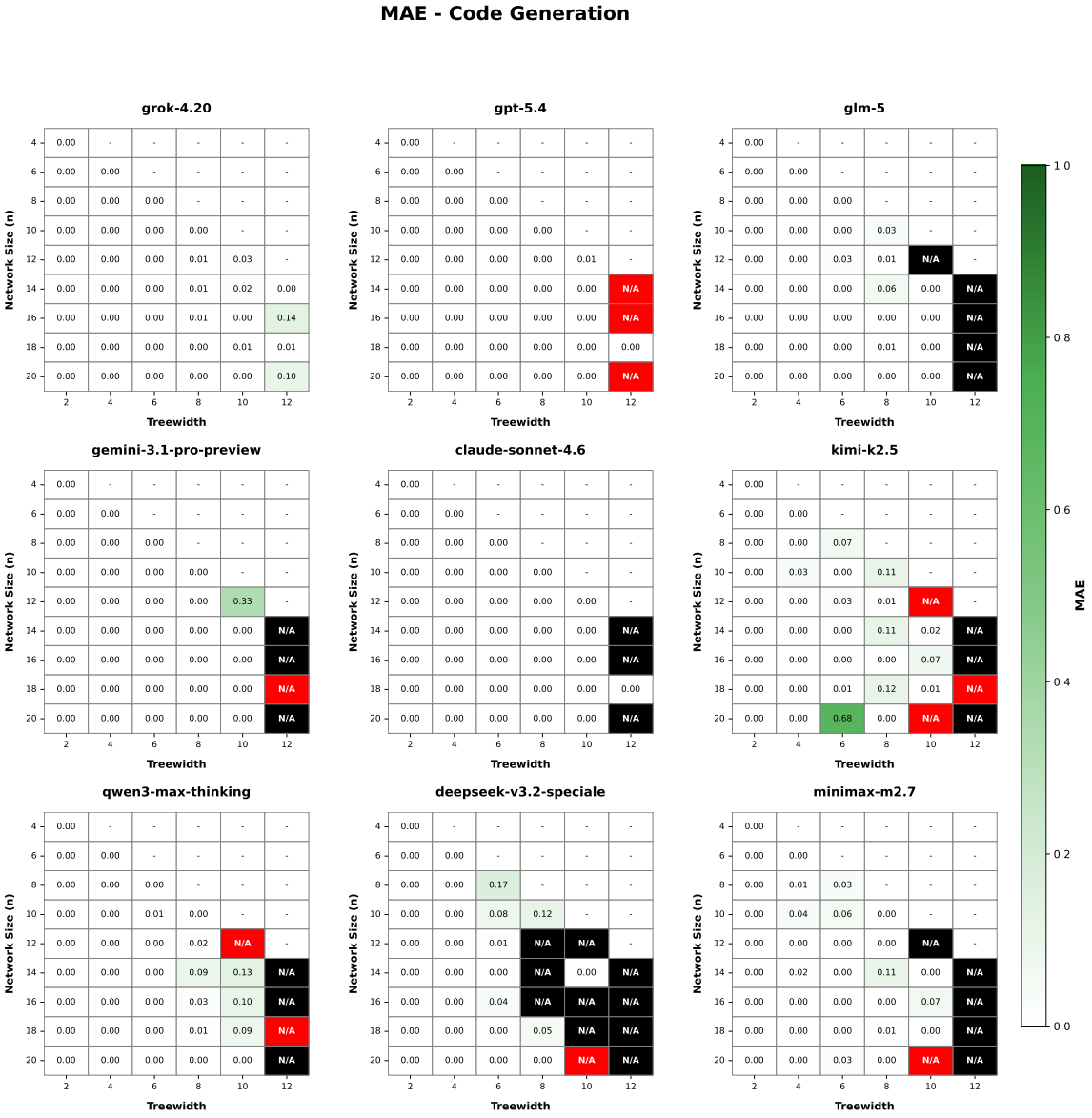


Figure 10: MAE by total factor size bin (y-axis) and treewidth (x-axis) in the code-generation protocol, using the same graph-level and query-level complexity axes as the code-generation accuracy analysis. Subplots correspond to models; darker green indicates higher error. White "–" cells denote missing query combinations, black "N/A" cells denote context-limited settings, and red "N/A" cells denote settings with no answerable queries.

Appendix G. Preliminary Analysis of Generated Code

This appendix examines the generated Python code. First, we classify failures to answer. Second, we compare BN-library and plain or general-purpose Python implementations over answerable queries, i.e., queries for which the model produces a valid, extractable numerical answer. Third, we present a matched qualitative analysis of 2 programs that solve the same BN query: a plain-Python enumeration generated by GPT-5.4 and a `pgmpy`-based solution generated by Gemini 3.1 Pro.

G.1. Failure modes

Of 3,546 code-generation queries that fit within the relevant model's context limit, 2,758 (77.8%) executed and printed a parseable number, while 788 (22.2%) did not. Table 5 classifies each of the 788 failures using the recorded exception type and message. Only 21 failures (2.7% of failures) are syntax or indentation errors. Most failures occur because the response contains no executable code block, violates the single-number output contract, or constructs an invalid BN representation. The latter category includes mismatched CPT shapes or cardinalities, missing or inconsistent graph parents, invalid state labels, and missing CPT/dictionary entries.

Execution alone does not establish semantic correctness. Of the 2,758 programs that printed a number, 2,643 (95.8%) were correct within the benchmark tolerance and 115 were not; 2 returned values outside $[0, 1]$. A static check also found 20 answerable programs whose final print argument was a numeric literal rather than a computed expression, only 6 of which were correct.

The compatibility rewrite described in Section 5.1 mapped imports of the removed `pgmpy` classes `BayesianModel` and `BayesianNetwork` to aliases of `DiscreteBayesianNetwork`. It was applied to 1,359 of the 3,546 context-supported generations (38.3%) and did not alter graph, CPT, query, or inference code.

Table 5: Failure modes among the 788 context-supported code generations that did not produce a parseable probability. Percentages use these 788 failures as the denominator. The categories are mutually exclusive.

Observed failure mode	**Count**	**Share**
No executable code block	458	58.1%
Invalid or missing single-number output	72	9.1%
BN/CPT/state or data-structure construction	160	20.3%
Python syntax or indentation	21	2.7%
Generated import or library-API error	15	1.9%
Sandbox-rejected operation	18	2.3%
Other Python runtime error	32	4.1%
Execution-environment mismatch (`pyAgrum`)	12	1.5%
Total	788	100.0%

G.2. Implementation modes

Table 6 shows that 8 of the 9 models used both `pgmpy` and plain or general-purpose Python in at least one answerable generation; GPT-5.4 was the exception, using no BN-specific library in all 383 of its answerable generations. The preferred mode varied substantially across models: for example, Gemini 3.1 Pro used `pgmpy` for 88.5% of its answerable generations, whereas Kimi K2.5 used it for only 7.8%. No answerable solution used `pyAgrum`, consistent with the execution-environment issue described above.

Accuracy was roughly similar between the 2 modes for most models. Among the 8 models that used both, the absolute difference in conditional accuracy was at most 6.3 percentage points for 7 models; the larger exception was MiniMax M2.7, with a 9.4-point difference. We did not analyze, however, whether a model tended to select one implementation mode for more difficult queries.

Table 6: Implementation mode and conditional accuracy among answerable code-generation queries. "No BN library" includes standard-library and general-purpose Python solutions.

Model	Answerable	pgmpy	Accuracy	No BN library	Accuracy
Claude Sonnet 4.6	326	178	100.00%	148	100.00%
DeepSeek V3.2-Speciale	192	104	97.12%	88	93.18%
Gemini 3.1 Pro	349	309	98.38%	40	100.00%
GLM-5	246	182	97.80%	64	92.19%
GPT-5.4	383	0	—	383	99.22%
Grok 4.20	367	271	97.79%	96	94.79%
Kimi K2.5	333	26	96.15%	307	89.90%
MiniMax M2.7	232	30	100.00%	202	90.59%
Qwen3-Max-Thinking	330	69	95.65%	261	90.80%

G.3. Matched generated-code example

We consider the query

$$P(V5 = s0 \mid V4 = s0), \tag{1}$$

from the 8-variable network in Figure 11. The network has achieved treewidth 6; after conditioning and pruning, the query has induced width 5 and total factor size 66. Its ground-truth probability is 0.622271. Both GPT-5.4 and Gemini 3.1 Pro print 0.622, which is correct under our requested precision (we ask for 3 digits).

The following parallel listings show the 2 generated programs side by side. Presentation-only whitespace, line wrapping, and quote style have been normalized; the programs' abstract syntax and numeric contents are unchanged. Both programs recover the relevant ancestral subnetwork and omit $V6$ and $V7$, whose conditional distributions sum out because they are unobserved descendants of the target and evidence variables.

GPT-5.4: plain Python

```
from itertools import product

states = (0, 1)  # 0 -> s0, 1 -> s1

P_V1 = {0: 0.3483, 1: 0.6517}
P_V0 = {0: 0.9858, 1: 0.0142}
```

Gemini 3.1 Pro: pgmpy

```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the network structure. We only need variables up to V5,
      as V6 and V7
# are unobserved descendants.
model = BayesianNetwork(
```

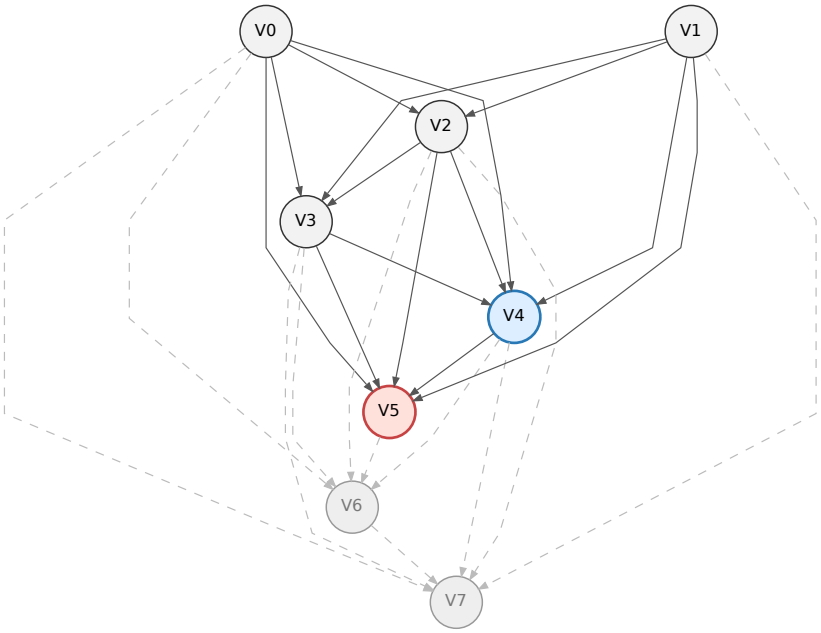


Figure 11: Network for the matched code example. The evidence variable $V4$ is blue and the target $V5$ is red. Gray nodes and dashed incoming edges mark $V6$ and $V7$, unobserved descendants that can be pruned for this query.

```
# CPTs store P(child = s0 | parents) in the parent order shown in
      the tables.
P_V2_s0 = dict(
    zip(
        product(states, states),
        [0.3599, 0.9923, 0.3922, 0.7487],
    )
)  # (V1, V0)

P_V3_s0 = dict(
    zip(
        product(states, states, states),
        [
            0.9704,
            0.2747,
            0.7502,
            0.2351,
            0.1909,
            0.8194,
            0.8314,
            0.7082,
        ],
    )
)  # (V1, V2, V0)

P_V4_s0 = dict(
    zip(
        product(states, states, states, states),
        [
            0.9599,
            0.2178,
            0.8318,
            0.8445,
            0.1742,
            0.3989,
            0.7755,
            0.7942,
            0.7204,
            0.8599,
            0.7074,
```

```
    [
        ("V1", "V2"),
        ("V0", "V2"),
        ("V1", "V3"),
        ("V2", "V3"),
        ("V0", "V3"),
        ("V1", "V4"),
        ("V2", "V4"),
        ("V3", "V4"),
        ("V0", "V4"),
        ("V1", "V5"),
        ("V2", "V5"),
        ("V3", "V5"),
        ("V4", "V5"),
        ("V0", "V5"),
    ]
)

# Define CPDs.
cpd_v1 = TabularCPD(variable="V1", variable_card=2,
                    values=[[0.3483], [0.6517]])
cpd_v0 = TabularCPD(variable="V0", variable_card=2,
                    values=[[0.9858], [0.0142]])

cpd_v2 = TabularCPD(
    variable="V2",
    variable_card=2,
    values=[
        [0.3599, 0.9923, 0.3922, 0.7487],
        [0.6401, 0.0077, 0.6078, 0.2513],
    ],
    evidence=["V1", "V0"],
    evidence_card=[2, 2],
)

cpd_v3 = TabularCPD(
    variable="V3",
    variable_card=2,
    values=[
        [0.9704, 0.2747, 0.7502, 0.2351,
```

```
            0.4862,
            0.3454,
            0.7247,
            0.9349,
            0.1112,
        ],
    )
)  # (V1, V2, V3, V0)

P_V5_s0 = dict(
    zip(
        product(states, states, states, states, states),
        [
            0.5443,
            0.6104,
            0.9824,
            0.5704,
            0.3041,
            0.6271,
            0.7677,
            0.2150,
            0.1930,
            0.3567,
            0.5140,
            0.7980,
            0.1338,
            0.4730,
            0.3542,
            0.8259,
            0.7965,
            0.7003,
            0.5188,
            0.2136,
            0.7290,
            0.0387,
            0.3190,
            0.7567,
            0.7523,
            0.4301,
            0.0078,
            0.3729,
            0.7402,
            0.6378,
            0.9435,
            0.6333,
        ],
    )
)  # (V1, V2, V3, V4, V0)


def p_state(p_s0, state):
    return p_s0 if state == 0 else 1.0 - p_s0


num = 0.0
den = 0.0

# For the denominator P(V4=s0), the unobserved descendants V5, V6,
      and V7
# sum out.
for v1, v0, v2, v3 in product(states, states, states, states):
    p = (
        P_V1[v1]
        * P_V0[v0]
        * p_state(P_V2_s0[(v1, v0)], v2)
        * p_state(P_V3_s0[(v1, v2, v0)], v3)
    )
    p_v4 = P_V4_s0[(v1, v2, v3, v0)]  # P(V4=s0 | ...)
    den += p * p_v4
    num += p * p_v4 * P_V5_s0[(v1, v2, v3, 0, v0)]

print(f"{num / den:.3f}")
```

```
         0.1909, 0.8194, 0.8314, 0.7082],
        [0.0296, 0.7253, 0.2498, 0.7649,
         0.8091, 0.1806, 0.1686, 0.2918],
    ],
    evidence=["V1", "V2", "V0"],
    evidence_card=[2, 2, 2],
)

cpd_v4 = TabularCPD(
    variable="V4",
    variable_card=2,
    values=[
        [0.9599, 0.2178, 0.8318, 0.8445,
         0.1742, 0.3989, 0.7755, 0.7942,
         0.7204, 0.8599, 0.7074, 0.4862,
         0.3454, 0.7247, 0.9349, 0.1112],
        [0.0401, 0.7822, 0.1682, 0.1555,
         0.8258, 0.6011, 0.2245, 0.2058,
         0.2796, 0.1401, 0.2926, 0.5138,
         0.6546, 0.2753, 0.0651, 0.8888],
    ],
    evidence=["V1", "V2", "V3", "V0"],
    evidence_card=[2, 2, 2, 2],
)

cpd_v5 = TabularCPD(
    variable="V5",
    variable_card=2,
    values=[
        [0.5443, 0.6104, 0.9824, 0.5704,
         0.3041, 0.6271, 0.7677, 0.2150,
         0.1930, 0.3567, 0.5140, 0.7980,
         0.1338, 0.4730, 0.3542, 0.8259,
         0.7965, 0.7003, 0.5188, 0.2136,
         0.7290, 0.0387, 0.3190, 0.7567,
         0.7523, 0.4301, 0.0078, 0.3729,
         0.7402, 0.6378, 0.9435, 0.6333],
        [0.4557, 0.3896, 0.0176, 0.4296,
         0.6959, 0.3729, 0.2323, 0.7850,
         0.8070, 0.6433, 0.4860, 0.2020,
         0.8662, 0.5270, 0.6458, 0.1741,
         0.2035, 0.2997, 0.4812, 0.7864,
         0.2710, 0.9613, 0.6810, 0.2433,
         0.2477, 0.5699, 0.9922, 0.6271,
         0.2598, 0.3622, 0.0565, 0.3667],
    ],
    evidence=["V1", "V2", "V3", "V4", "V0"],
    evidence_card=[2, 2, 2, 2, 2],
)

model.add_cpds(cpd_v1, cpd_v0, cpd_v2, cpd_v3, cpd_v4, cpd_v5)

# Perform Variable Elimination.
infer = VariableElimination(model)

# Query P(V5=s0 | V4=s0); state s0 corresponds to index 0.
result = infer.query(variables=["V5"], evidence={"V4": 0})

print(f"{result.values[0]:.3f}")
```

Plain-Python program. GPT-5.4 stores the $s0$ probability for each CPT column and reconstructs the complementary $s1$ probability with `p_state`. It then enumerates the $2^4 = 16$ assignments to $(V1, V0, V2, V3)$. For each assignment it evaluates the ancestral joint factor, accumulates $P(V4 = s0)$ in `den`, and accumulates $P(V5 = s0, V4 = s0)$ in `num`. The final division therefore implements the conditional-probability definition directly.

Importantly, this is not blind enumeration of the full joint distribution. With the evidence $V4 = s0$ fixed, enumerating every other binary variable would require $2^7 = 128$

assignments. GPT-5.4 instead recognizes that the unobserved descendants $V6$ and $V7$ sum to one and omits both their assignments and their CPTs. It also sums out $V5$ analytically in the denominator and selects the requested $V5 = s0$ CPT entry in the numerator. Consequently, only the four hidden ancestors remain, reducing the enumeration domain by a factor of eight; the same loop also supplies both numerator and denominator.

Its effective arithmetic workload is obtained by expanding the 16 loop iterations. The program executes 32 additions, 96 multiplications, 16 subtractions to reconstruct $s1$ probabilities, and 1 final division, for a total of 145 scalar arithmetic operations. Under the benchmark's VE convention, which counts dense scalar additions and multiplications but not divisions or complement construction, the query requires 31 additions and 92 multiplications. The directly comparable addition-plus-multiplication totals are therefore 128 for GPT-5.4 and 123 for VE.

Library-based program. Gemini 3.1 Pro reconstructs the relevant DAG, represents each CPT as a `TabularCPD` with the correct parent order and binary cardinalities, and calls `VariableElimination.query` with target `V5`, evidence `V4=0`, and target-state index 0.

Appendix H. Treewidth and Total Factor Size Effects Controlling for Prompt Length and Network Size

This appendix provides additional details for Section 6.3. For each protocol, we fit pooled logistic regressions on supported examples only, excluding examples that exceeded the context length limit. The dependent variables were answerability (whether a valid numerical answer could be extracted) and accuracy (whether the answer was within $\tau = 0.01$ of the ground truth, with invalid answers counted as incorrect). In all regressions, treewidth, input tokens, network size n, and total factor size were standardized to zero mean and unit variance, and model fixed effects were included:

$$\operatorname{logit} \Pr(Y = 1) = \beta_0 + \beta_{\text{tw}} z(\text{tw}) + \beta_{\text{tok}} z(\text{input_tokens}) + \beta_n z(n) + \beta_{\text{fs}} z(\text{factor_size}) + \gamma_{\text{model}}. \tag{2}$$

Here, Y is either the answerability or accuracy indicator, $z(\cdot)$ denotes standardization to zero mean and unit variance, total factor size is the sum of the intermediate factor sizes generated by variable elimination plus the final target-factor size, and γ_{model} denotes model fixed effects, treating the model (e.g., "GPT-5.4") as a categorical variable so the regression does not mix overall differences between LLMs with the effects of treewidth, input length, network size, or query total factor size.

The treewidth and input-token predictors were moderately correlated in both protocols ($r = 0.555$ for raw reasoning and $r = 0.556$ for code generation), as were total factor size and treewidth ($r = 0.343$ for raw reasoning and $r = 0.344$ for code generation). This analysis therefore tests whether graph-level and query-level complexity remain predictive after controlling for prompt length, network size, and model identity. Section 6.3 reports the pooled coefficients, and Tables 7 and 8 report per-model fits. Because the predictors are standardized, odds ratios (OR) correspond to a one-standard-deviation increase in the associated variable, and negative coefficients (β) indicate lower odds of success as the predictor increases. Across all four regressions, treewidth and total factor size have negative and statistically significant effects, with total factor size especially predictive of accuracy. By contrast, the coefficient on standardized network size is smaller and less consistent, suggesting that once treewidth, prompt length, and query-complexity burden are accounted for, node count contributes comparatively little additional explanatory power. We report AUC and McFadden pseudo-R^2 as goodness-of-fit summaries.

H.1. Per-Model Regressions

We also fit the augmented logistic regression separately for each model, using the same standardized predictors as in the pooled analysis and omitting model fixed effects. For raw reasoning, some per-model fits exhibit quasi-complete separation, undefined fit diagnostics, or very large coefficient magnitudes, so Table 7 reports only the models for which both the answerability and accuracy fits were stable. The excluded raw-reasoning models are Claude Sonnet 4.6, DeepSeek V3.2-Speciale, Grok 4.20, Kimi K2.5, MiniMax M2.7, and Qwen3-Max-Thinking. In the code-generation protocol, all per-model fits were stable enough to report (Table 8). These descriptive regressions show that the aggregate effects mask heterogeneous behavior: total factor size is a strong negative predictor of code-generation accuracy

for Claude Sonnet 4.6, Gemini 3.1 Pro, Grok 4.20, Kimi K2.5, MiniMax M2.7, and Qwen3-Max-Thinking, while GPT-5.4 is affected more clearly by input length than by total factor size in this protocol.

Table 7: Stable per-model logistic regressions for the raw-reasoning protocol. Coefficients are standardized log-odds; AUC and McFadden pseudo-R^2 summarize goodness of fit. Models with unstable per-model fits are omitted.

Model	**Outcome**	**Rows**	**Rate**	β_{tw}	β_{tok}	β_n	β_{fs}	**AUC**	**McFadden R^2**
Gemini 3.1 Pro	Answerability	402	0.908	-5.714^{***}	0.007	2.615^{***}	-0.104	0.976	0.636
GLM-5	Answerability	379	0.770	-0.466^{*}	1.139	-0.706^{***}	-0.177	0.730	0.102
GPT-5.4	Answerability	434	0.825	-0.696	-0.664^{**}	-0.191	-2.657^{***}	0.963	0.597
Gemini 3.1 Pro	Accuracy	402	0.789	-3.147^{***}	-0.343	1.911^{***}	-3.451^{***}	0.970	0.625
GLM-5	Accuracy	379	0.578	-0.746^{***}	0.428	-0.232	-2.248^{***}	0.793	0.184
GPT-5.4	Accuracy	434	0.813	-0.636	-0.914^{***}	-0.024	-3.190^{***}	0.971	0.647

Significance codes: $^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$.

Table 8: Per-model logistic regressions for the code-generation protocol. Coefficients are standardized log-odds; AUC and McFadden pseudo-R^2 summarize goodness of fit.

Model	**Outcome**	**Rows**	**Rate**	β_{tw}	β_{tok}	β_n	β_{fs}	**AUC**	**McFadden R^2**
Claude Sonnet 4.6	Answerability	403	0.809	-1.076^{***}	-0.195	0.490^{*}	-1.807^{***}	0.848	0.320
DeepSeek V3.2-Speciale	Answerability	310	0.619	-0.427	-2.347	0.366^{**}	-0.119	0.655	0.062
Gemini 3.1 Pro	Answerability	403	0.866	-2.559^{***}	-0.158	1.215^{**}	-1.487^{***}	0.920	0.497
GLM-5	Answerability	379	0.649	-0.543^{**}	-0.673	-0.140	-0.435	0.717	0.106
GPT-5.4	Answerability	434	0.882	-0.353	-0.864^{***}	-0.661^{*}	-0.175	0.917	0.481
Grok 4.20	Answerability	434	0.846	-0.485^{*}	-0.017	-0.361	-0.771^{***}	0.730	0.211
Kimi K2.5	Answerability	402	0.828	0.199	-2.983^{***}	-0.740^{**}	-0.812^{**}	0.796	0.315
MiniMax M2.7	Answerability	379	0.612	-1.383^{***}	-0.692	-0.055	-0.805	0.841	0.295
Qwen3-Max-Thinking	Answerability	402	0.821	-1.153^{***}	1.126	0.013	-0.223	0.507	0.107
Claude Sonnet 4.6	Accuracy	403	0.809	-1.076^{***}	-0.195	0.490^{*}	-1.807^{***}	0.848	0.320
DeepSeek V3.2-Speciale	Accuracy	310	0.590	-0.656^{*}	-1.861	0.467^{***}	-0.146	0.684	0.089
Gemini 3.1 Pro	Accuracy	403	0.854	-2.510^{***}	-0.856^{*}	1.668^{***}	-1.874^{***}	0.934	0.579
GLM-5	Accuracy	379	0.625	-0.699^{**}	-0.756	-0.072	-0.266	0.736	0.123
GPT-5.4	Accuracy	434	0.876	-0.403	-0.873^{***}	-0.527	-0.170	0.923	0.479
Grok 4.20	Accuracy	434	0.820	-0.440^{*}	-0.060	-0.289	-1.521^{***}	0.771	0.304
Kimi K2.5	Accuracy	402	0.749	-0.289	-1.990^{***}	-0.032	-1.522^{***}	0.798	0.288
MiniMax M2.7	Accuracy	379	0.562	-1.232^{***}	-1.270	0.052	-1.555^{*}	0.844	0.302
Qwen3-Max-Thinking	Accuracy	402	0.754	-0.986^{**}	0.045	0.143	-3.115^{***}	0.663	0.349

Significance codes: $^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$.