CODOKU: RENEWABLE PROGRAM-REASONING CHALLENGES FOR FRONTIER CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing program-reasoning benchmarks ask large language models to predict a program's behavior on a given input. Coding agents break two assumptions on which these benchmarks rest: an agent can recover the answer by executing the program instead of reasoning about it, and fixed task sets drawn from existing programs are increasingly exposed to contamination, yet costly to renew. We introduce *Codoku* (*code sudoku*), a renewable benchmark in which a solver fills typed cells in a partial program to satisfy global static and dynamic constraints, such as a prescribed control-flow graph and execution path. Because a partial program cannot be executed and valid fillings are sparse in an exponentially large space of interdependent choices, neither tool use nor enumeration can substitute for program reasoning. Puzzles are synthesized from scratch via semantic reification, so fresh puzzles of controllable complexity can be generated on demand, each with a witness that guarantees solvability. We evaluate five frontier models on 300 puzzles through a coding agent free to use any tool within a fixed budget. Small puzzles already challenge open-weight models, whereas even proprietary models solve only about half of the large ones. Codoku thus offers a renewable testbed for program reasoning that can keep pace with rapidly improving coding agents.

1 INTRODUCTION

The ability to reason about programs, which we refer to as *program reasoning*, is fundamental to large language models (LLMs) on coding tasks. A substantial line of benchmarks evaluates this ability through the relation among a program, its inputs, and its execution: given a complete program and a concrete input, an LLM is asked to predict outputs, intermediate states, branch outcomes, execution paths, or unexpected behaviors (Gu et al., 2024; Chen et al., 2025; Li et al., 2025; Xie et al., 2025; Becker et al., 2026; Spiess et al., 2026). These benchmarks have enabled meaningful comparisons among LLMs and have supported real progress in program reasoning. They were designed, however, for *bare models*. Today, LLMs increasingly operate inside *coding agents*, *i.e.*, harnesses that equip them with shells, interpreters, compilers, debuggers, fuzzers, and other tools. Transplanting these benchmarks into this setting is more than a change of harness: it invalidates two critical assumptions on which they rest, one concerning their *format* and the other their *supply*.

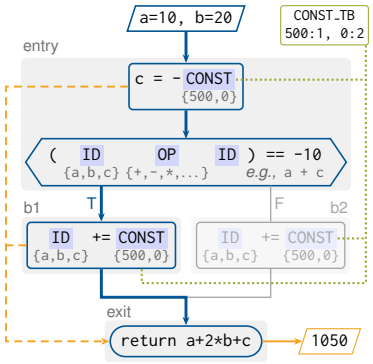
Tool access decouples task success from program reasoning. Predicting what a program does serves as a proxy for reasoning about it only as long as the predictor cannot execute the program; an agent, however, can. Every quantity that these benchmarks measure, such as outputs and intermediate states, is recoverable by execution, instrumentation, or debugging, so a task can be solved with a few tool calls and without any understanding of the program or its semantics. Execution-prediction benchmarks such as CruxEval (Gu et al., 2024) and REval (Chen et al., 2025) remain valuable for evaluating bare models, but their task format is no longer suitable for coding agents.

Nonrenewable benchmarks cannot supply fresh challenges. All program-reasoning benchmarks that we are aware of draw their tasks from existing programs: textbook exercises, programming contests, and GitHub repositories. These tasks are either created by hand for quality or mined automatically for scale. Both strategies yield valuable tasks, yet both freeze a benchmark at publication time while the LLMs under evaluation keep evolving. Because the originating programs and their traces are public, each training cycle increases the likelihood that they have been absorbed. Web access aggravates the problem further, as an agent may simply retrieve the source program or even a published solution.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```python
#@CFG_EDGE: entry -> b1, b2
#@CFG_EDGE: b1, b2 -> exit
#@CONST_TB: 500:1, 0:2
#@INOUT_EX: codoku(10,20) == 1050
#@EXE_PATH: entry -> b1 -> exit
def codoku(a: int, b: int):
  #@CFG_BLOCK: entry
  c: int = -CONST
  if (ID OP ID) == -10:
    #@CFG_BLOCK: b1
    ID += CONST
  else:
    #@CFG_BLOCK: b2
    ID += CONST
  #@CFG_BLOCK: exit
  return a + 2 * b + c
```

(a) Puzzle



(b) Core Structure

```python
#@CFG_EDGE: entry -> b1, b2
#@CFG_EDGE: b1, b2 -> exit
#@CONST_TB: 500:1, 0:2
#@INOUT_EX: codoku(10,20) == 1050
#@EXE_PATH: entry -> b1 -> exit
def codoku(a: int, b: int):
  #@CFG_BLOCK: entry
  c: int = -0
  if (c - a) == -10: # or (a - b)
    #@CFG_BLOCK: b1
    b += 500
  else:
    #@CFG_BLOCK: b2
    a += 0            # or b, or c
  #@CFG_BLOCK: exit
  return a + 2 * b + c
```

(c) Six Solutions

Figure 1: **Valid solutions are sparse**. (a) A toy puzzle with eight typed cells. (b) Its core structure, with one node per statement. On input `codoku(10,20)`, execution must follow `entry -> b1 -> exit` (blue), so `b2` never runs (grey). With three choices per `ID`, two per `CONST`, and 25 for `OP`, there are $3^4 \times 2^3 \times 25 = 16{,}200$ candidate fillings. Global constraints couple distant cells: the return value must be `1050`, which ties the `return` to `c = -CONST` and to the update in `b1` (orange), and the three `CONST` cells share one constant table (green). (c) The six valid fillings: the branch condition admits two choices and the `ID` in the unexecuted `b2` admits three; all other cells are forced.

The natural remedy, renewal, is costly: as LLMs become more capable, renewing these benchmarks at comparable quality and quantity requires substantial effort.

Codoku: renewable challenges for frontier coding agents. We introduce a new kind of program-reasoning challenge, *codoku* (short for *code sudoku*). Like sudoku, a codoku puzzle asks a solver to fill *cells* whose choices are linked by *global constraints*. Whereas sudoku constrains the digits in each row, column, and block, codoku constrains a program's static structure and dynamic behavior.

Figure 1 presents a toy codoku puzzle and its six solutions, and Section A.6 presents a full-scale puzzle. A solution must fill every `ID`, `CONST`, and `OP` cell such that the completed program

1. is accepted by the Python compiler;
2. matches the control-flow graph (CFG) declared by the `#@CFG_EDGE` and `#@CFG_BLOCK` annotations;
3. uses only constants listed in `#@CONST_TB`, each as often as specified (`500` once and `0` twice); and
4. on input `codoku(10,20)`, returns `1050` (`#@INOUT_EX`) along the execution path (EP) in `#@EXE_PATH`.

The first three constraints are static, whereas the fourth constrains the program's dynamic behavior.

Reasoning by necessity. Codokus make program reasoning unavoidable, even for agents equipped with tools. First, a codoku puzzle has no behavior to observe: being a partial program, it neither compiles nor runs, so execution, instrumentation, and debugging cannot reveal a valid filling. Second, its search space grows exponentially with the number of cells, whereas valid solutions are sparse, which renders random guessing and exhaustive enumeration impractical, if not infeasible; even the toy puzzle in Figure 1 admits 16,200 fillings, only six of which are valid. To fill a cell, a solver therefore *must* characterize the program states that can reach the corresponding program point, and then propagate the consequences of its choice forward along the prescribed execution path and backward from the required return value. Because a locally plausible choice may surface as a violation only several statements or loop iterations later, cells cannot be resolved independently.

Renewable by construction. To generate codokus, we build on *semantic reification* (Chopra et al., 2026), a recent technique from the programming-languages community that synthesizes programs from scratch to satisfy prescribed semantic constraints. Given a CFG, an input, and an EP, our synthesizer constructs a terminating program $P^\star$ that follows the path on the input. Masking $P^\star$ with typed cells yields a puzzle, and $P^\star$ itself serves as a *witness* that the puzzle has at least one valid solution. Solutions, however, are verified against the global constraints rather than against the witness: our solution checker parses a submitted filling, checks the constraints one by one, and accepts the filling if it satisfies all of them. For the puzzle in Figure 1, the checker thus accepts all six solutions.

Because puzzles are synthesized from scratch, they are independent of existing programs and renewable by construction, which addresses the supply problem: fresh instances can always be drawn after a model is trained, reducing the risk that the exact programs and solutions have been seen during training or can be retrieved from the web. The synthesizer exposes parameters that control puzzle complexity, including program size and the number of loop iterations. Based on these parameters, we define three generation profiles: a `small` codoku encodes a small straight-line function with a short execution path, a `medium` codoku a function with shallow loops and a moderately long execution path, and a `large` codoku a large branching function with nested loops. Rather than assuming that these profiles directly determine difficulty, we measure how they affect agent performance.

Evaluating frontier agents. Codokus are designed for both bare models and coding agents, and we do not restrict how an agent solves them: an agent may use any available tool to execute candidate fillings, invoke the solution checker, inspect failures, and implement its own search procedure in any programming language. We therefore measure whether an agent finds a valid solution within a fixed resource budget, regardless of the method it uses. We evaluate five frontier models with the Pi coding agent (Pi, 2026) on 300 codoku puzzles, 100 from each generation profile. Codoku puzzles challenge coding agents: even on small puzzles, solve rates range from 39% to 77%, and the highest solve rate on large puzzles is 54%. Every model solves fewer large puzzles than small ones, and resource use grows with profile scale. Tool-use trajectories further show that agents combine multiple strategies to infer local program states to solve puzzles. Together, these results establish our contribution: Codoku, a renewable testbed for program reasoning that can keep pace with rapidly improving coding agents.

2 Codoku Construction

Codoku formulates program reasoning as constraint satisfaction over a masked program. Constructing a puzzle therefore requires producing a partial program, together with static and dynamic semantic constraints, that jointly meet four requirements:

1. every puzzle admits at least one witness solution by construction (*solvable*);
2. its solution space is large while its valid solutions are few (*challenging*);
3. a candidate solution can be verified objectively in polynomial time (*verifiable*);
4. fresh puzzles can always be drawn on demand (*renewable*).

Figure 2 illustrates the pipeline on the puzzle of Figure 1: witness synthesis (Section 2.1), puzzle construction (Section 2.2), and solution verification (Section 2.3).

2.1 Witness Synthesis

Most program generators are syntax-guided: they emit syntactically correct programs but offer no guarantees about runtime behavior on a specific input, so their output may raise unhandled exceptions, enter infinite loops, or consist largely of dead code. For example, Csmith (Yang et al., 2011) guarantees semantic correctness, yet does not offer precise control over semantic properties such as loop behavior and execution length. Lacking such control, the resulting codoku puzzles would be too easy or too hard. We therefore build on *semantic reification* (Chopra et al., 2026), which synthesizes programs that satisfy prescribed static and dynamic semantic specifications.

Semantic reification. Let $g = (V, E)$ be a control-flow graph (CFG), where V denotes the set of basic blocks, among them a unique `entry` and a unique `exit` block, and $E \subseteq V \times V$ the set of directed control-flow edges. An execution path (EP) through g is a finite walk $\pi = [b_1, b_2, \ldots, b_{|\pi|}]$ from the entry to the exit, *i.e.*, $b_1 = \texttt{entry}$ and $b_{|\pi|} = \texttt{exit}$. From a given CFG g and EP π, semantic reification synthesizes a program $P^\star$ together with an input i and the corresponding output o:

$$\texttt{reify}(g, \pi) \rightsquigarrow (P^\star, i, o) \quad \textit{s.t.} \quad \big(\texttt{cfg}(P^\star) \cong g\big) \wedge \big(P^\star(i) \Downarrow o\big) \wedge \big(\texttt{ep}(P^\star, i) = \pi\big).$$

That is, $P^\star$'s CFG is isomorphic to g, executing $P^\star$ on i terminates deterministically with output o, and the sequence of basic blocks traversed by that execution is exactly π.

Witness synthesis. We instantiate semantic reification in four steps:

1. *CFG and EP creation*: We construct a connected CFG g by adding forward branches and loop back edges, then walk it to obtain the EP π.

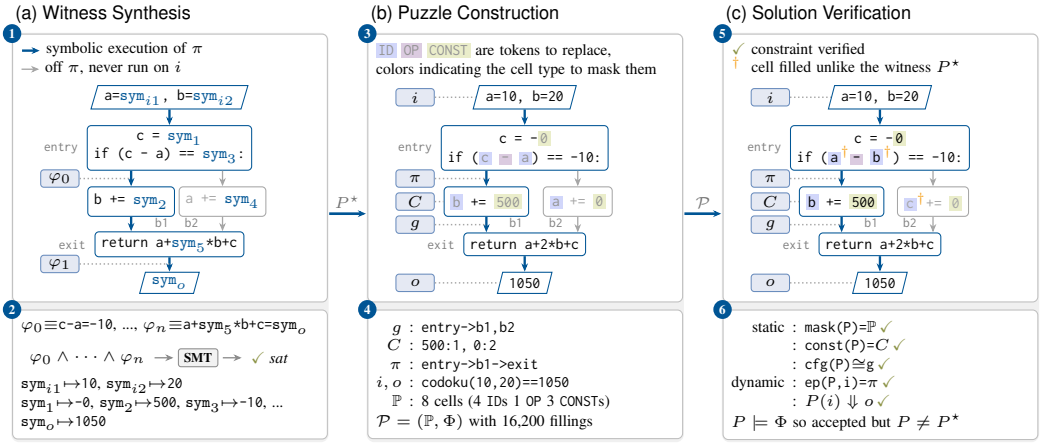


Figure 2: **Building a codoku puzzle and checking a solution** (for Figure 1). **(a)** Semantic reification populates a required CFG with symbolic inputs, intermediates, and output ❶, encodes the conditions imposed by the path π, and solves them in one SMT query ❷, returning a witness solution $P^\star$. **(b)** Masking replaces shaded tokens with typed cells ❸. The witness supplies the constraints: its graph gives g, its constants the constant table C, and its run on i the path π and output o ❹. **(c)** A candidate is checked in the same places ❺: first statically (cells, structure, constants), then dynamically by running it on i ❻. Checking against Φ, not $P^\star$, accepts valid fillings that differ from the witness.

2. *Statement population*: We populate every basic block with symbolic Python statements, seeding the blocks along π with statements over *symbolic* variables and filling those outside π with concrete statements. Off-path code is never executed on the input i, which Step 4 determines.
3. *Symbolic execution*: We symbolically execute the populated program along π, encoding the path, definedness, and computation conditions required to follow π as bit-vector constraints.
4. *Program concretization*: We solve these constraints with an SMT solver, obtaining concrete values for the input i, the output o, and every symbolic variable. Substituting each symbolic variable with its solved value yields the final program $P^\star$.

$P^\star$ is the witness solution from which we construct a codoku puzzle, while the CFG g, EP π, input i, and output o are retained as constraints for solution verification.

2.2 Puzzle Construction

Our puzzle synthesizer turns the witness program $P^\star$ into a codoku puzzle $\mathcal{P} = (\mathbb{P}, \Phi)$ in three steps: it (1) replaces selected tokens with typed cells to form the puzzle body $\mathbb{P}$, (2) collects the masked constants into a table, and (3) records the global constraints Φ as annotations.

Typed cells. The synthesizer preserves $P^\star$'s function signature. Within the function body, it selects eligible statements and replaces their maskable tokens with typed cells, yielding $\mathbb{P}$:

- `ID`: A local variable or parameter identifier such as `v1`.
- `FUNC`: A function name available in the puzzle such as `sat_add`.
- `CONST`: A numeric constant, either an integer or a floating-point number such as `500`.
- `OP`: A supported operator or conditional-expression keyword such as `+`.
- `CTRL`: A control-flow keyword, either `break` or `continue`.
- `LABEL`: A destination basic-block label such as `b1`.

Not every puzzle contains all six kinds. A cell's type restricts its fillings: `ID` and `FUNC` cells admit only variables and functions declared in the puzzle, respectively, and `CONST` cells only table constants.

Constant table. The synthesizer collects the constants replaced by `CONST` into a table $C = \{(c_1, n_1), (c_2, n_2), \ldots, (c_{|C|}, n_{|C|})\}$, where c_j is a constant and n_j its required number of occurrences. Constants are distinguished by value and by type, so that the integer `1` and the floating-point `1.0` are distinct. A valid solution must assign each c_j to exactly n_j constant cells and may use no constant outside the table. Spending a constant in one cell thus reduces its remaining occurrences

elsewhere, which prevents a solver from satisfying statements independently. This budget, inspired by sudoku, rules out the arbitrary literals that would otherwise make individual cells easy to satisfy.

Global constraints. The synthesizer records global constraints $\Phi = (g, C, i, \pi, o)$ as annotations alongside the puzzle: `#@CFG_{BLOCK,EDGE}` declare basic blocks and control-flow edges, `#@CONST_TB` displays the constant table, and `#@INOUT_EX` and `#@EXE_PATH` give the input-output example and the required EP. Cell types in $\mathbb{P}$, such as `ID`, further limit the admissible choices. These constraints serve complementary roles: the CFG constrains the program's static structure, while the EP and the output constrain its runtime behavior. Together they define an explicit, constrained program-reasoning task, rather than an attempt to infer an unspecified program from input-output examples.

2.3 Solution Verification

The witness $P^\star$ establishes that at least one valid filling exists, but a solver need not recover its particular choices. The checker accordingly evaluates a candidate against the global constraints rather than against $P^\star$, and accepts every satisfying filling, *i.e.*, $\texttt{valid_sols}(\mathcal{P}) = \{P \mid P \models \Phi\}$, where

$$P \models \Phi \iff \big(\texttt{mask}(P) = \mathbb{P}\big) \wedge \big(\texttt{const}(P) = C\big) \wedge \big(\texttt{cfg}(P) \cong g\big) \wedge \big(P(i) \Downarrow o\big) \wedge \big(\texttt{ep}(P, i) = \pi\big).$$

Static constraints. The checker first confirms that every cell is filled and that the candidate parses and compiles. It then traverses the candidate's abstract syntax tree (AST) to perform four checks:

- *Cell and structure preservation*: Every cell must be filled with a token of its type, and all unmasked code must remain unchanged. The checker masks P by the procedure that produced the typed cells of $P^\star$ (Section 2.2); the result must be identical to the puzzle at the AST level, comments ignored. Formatting changes are thus permitted, whereas altering an unmasked statement or inserting new code, such as replacing the puzzle body with a hardcoded return value, is not.
- *Constant consistency*: Cell constants must match the constant table in value, type, and count.
- *Declaration consistency*: Variables and functions used to fill cells must be declared in the puzzle.
- *CFG consistency*: The CFG reconstructed from the candidate must match the declared CFG g.

Dynamic constraints. The checker executes the candidate on i and records the sequence of basic blocks it visits. The candidate must terminate with output o, and the recorded sequence must match the EP π exactly, including repeated visits. Output agreement alone is therefore insufficient: a candidate that returns o but skips a required block is invalid, as is one that follows π but returns a different value. A 5-second execution time limit further rejects candidates that do not finish in time.

Checker feedback. The checker stops at the first failed check and reports the violated constraint, for instance a missing or unexpected CFG edge, a mismatch in the execution path, or an incorrect constant count. Such feedback lets agents revise a candidate in response to a specific failure, yet it neither discloses the witness's choices nor requires agents to reproduce them.

2.4 Generation Profiles

The complexity of a puzzle can be measured in several respects, of which we consider two: search-space complexity and program complexity. As in sudoku, the search-space complexity of a puzzle is the number of assignments admissible to its cells, that is, the size of the Cartesian product of the cells' domains; it counts all permitted fillings, not only those that satisfy the puzzle constraints. Program complexity metrics, widely used in software engineering and programming languages to estimate how difficult a program is to understand, test, maintain, or modify, describe both static program structure and dynamic execution behavior, capturing properties that cell counts alone omit. Following existing literature (Peitek et al., 2021) and the metrics plugins popular for VS Code (Tamás, 2016), IntelliJ Platforms (BasLeijdekkers, 2015), and Eclipse (Beyer, 2009), we adopt for example the numbers of statements and variables, McCabe's cyclomatic complexity (McCabe, 1976), the length of execution paths, and the number of loop iterations, which distinguish code size from the amount of execution on the specified input. The synthesizer exposes both as parameters, from which we derive three profiles:

- **Small**: Generate small straight-line functions with short execution.
- **Medium**: Generate larger functions with shallow loops and moderate execution.
- **Large**: Generate even larger branching functions with nested loops and long execution.

Table 1: **Codokus challenge every evaluated agent, and larger profiles are more complex and generally harder**. The left table displays average puzzle properties per profile. The right table shows mutually exclusive run outcomes (Section 3.3): a run is *exhausted* when it reaches its time, cost, or request budget, and *failed* when it otherwise ends without a valid solution.

(a) Puzzle Complexity

Metric	Small	Medium	Large
search space			
Typed cells	58.54	134.65	366.49
Log10 space	71.99	182.26	540.03
static structure			
Lines of code	38.85	71.94	133.46
Halstead diff.	38.62	58.31	86.18
CFG nodes	4.77	6.58	9.93
CFG edges	5.47	8.91	14.60
Cyclo. compl.	2.70	4.33	6.67
Data dep. nodes	21.35	44.01	92.95
Data dep. edges	16.74	46.29	111.66
Data dep. degree	1.50	2.07	2.38
dynamic behavior			
EP length	5.85	9.71	15.29
Unique blocks	4.70	5.92	7.07
Repeated blocks	1.15	3.79	8.22
Loop iterations	1.12	2.52	4.09
input-output examples			
Examples	3.71	6.30	9.00

(b) Model Performance with the Pi Agent

Model	Profile	Solved	Exhausted	Failed
Claude Opus 5	***Total***	63%	33%	4%
	· *small*	77%	20%	3%
	· *medium*	62%	33%	5%
	· *large*	50%	45%	5%
GPT 5.6 Sol	***Total***	58%	36%	6%
	· *small*	67%	33%	0%
	· *medium*	53%	43%	4%
	· *large*	54%	33%	13%
GLM 5.3	***Total***	28%	58%	14%
	· *small*	50%	40%	10%
	· *medium*	22%	53%	25%
	· *large*	12%	80%	8%
Kimi K3	***Total***	29%	70%	1%
	· *small*	48%	51%	1%
	· *medium*	28%	72%	0%
	· *large*	11%	89%	0%
DPSK V4.1 Flash	***Total***	30%	64%	6%
	· *small*	39%	61%	0%
	· *medium*	29%	65%	6%
	· *large*	22%	66%	12%

Section 3.2 reports each profile's empirical complexity. A more complex puzzle is generally harder, but not always: a puzzle with a large search space may be easy if local constraints eliminate most choices. We therefore assess difficulty through agent success rates and solution costs (Section 3.3).

3 Evaluation

Because codokus are renewable by construction, our evaluation focuses on whether they are challenging for frontier LLMs operating through a coding agent. We address two research questions:

- **RQ1: Codoku characterization**. Do the small, medium, and large profiles yield puzzles of increasing search space and program complexity?
- **RQ2: Challenge for coding agents**. How successfully do current coding agents solve codokus within a fixed resource budget, and how does their success vary across the generation profiles?

3.1 Evaluation Setup

We generate 100 puzzles per profile, 300 in total. Each puzzle comes with a witness solution, which is unavailable to agents. We evaluate Claude Opus 5, GPT 5.6 Sol, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash, each with its thinking effort set to `high`. All models run in the Pi coding agent (Pi, 2026) under an identical configuration. The agent has access to the solution checker (Section 2.3) and all standard tools, and each agent is isolated in its own E2B micro VM with four CPU cores and 16 GB of memory. Every model attempts all 300 puzzles, for 1,500 runs in total.

Budget. For each puzzle, we grant the agent at most one hour, $15 in model usage, and 128 model requests. A run ends when the agent submits a solution or exhausts any of these budgets.

3.2 RQ1: Codoku Characterization

Every metric in Table 1a increases monotonically from the small to the medium to the large profile. The profiles scale both dimensions of complexity (Section 2.4): the search space and program complexity as reflected in static structure, dynamic behavior, and the number of input-output examples.

The profiles do not, however, merely lengthen the programs. From the small to the large profile, code size grows 3.4× but the number of typed cells 6.3×, so each line carries nearly twice as many

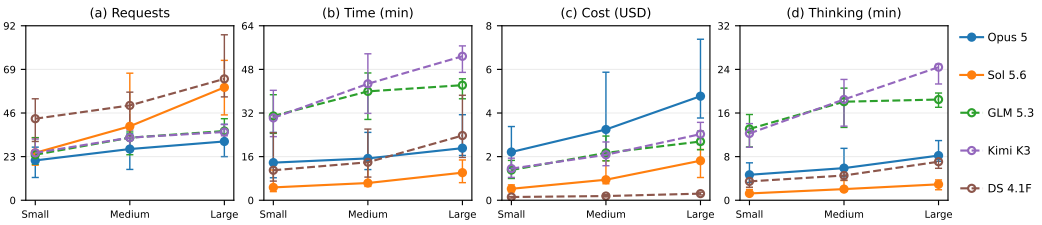


Figure 3: **Resource use rises with profile scale among solved runs**. Each plot reports the per-run median and interquartile range over solved puzzles.

cells; the average data-dependency degree rises by more than 50%, so each value takes part in more dependencies; and repeated block visits, which make up 20% of the execution path in small puzzles, exceed 53% of it in large ones. Larger puzzles therefore contain not only more decisions but also more relations among them. Execution paths grow mainly through iteration, so each decision propagates through more loop iterations. Larger profiles also supply more input-output examples, each both a further constraint on the filling and a further source of information (details in Section A.1).

Answer to RQ1. The realized complexity follows the intended ordering: the profiles scale both a solver's choices and the semantic dependencies that constrain them. They do so without repository-scale programs: even large puzzles average 133 lines of code, whereas even small puzzles span on the order of 10^{72} candidate fillings. RQ2 asks whether such compact puzzles challenge current agents.

3.3 RQ2: Challenge for Coding Agents

We classify each run into one of three mutually exclusive outcomes as shown in Table 1b. Additionally, a run is *failed* if it ends without a valid solution for any other reason: a safety refusal by the provider, an output-length error, an infrastructure failure, or the agent stopping without a valid completion.

Compact puzzles pose substantial challenges. No evaluated agent saturates codoku. Claude Opus 5 reaches the highest overall solve rate, followed by GPT 5.6 Sol, yet even Claude Opus 5 leaves almost a quarter of the small puzzles unsolved, although these average fewer than 40 lines of code. No agent solves more than 54% of the large ones. Substantial headroom thus remains at every profile, even though the agents can execute code, validate candidates, and program their own search.

A gap exist between proprietary and open-weight models. The two proprietary models solve roughly twice as many puzzles as the three open-weight ones, and the separation holds within every profile: both proprietary models outperform all three open-weight models on small, medium, and large puzzles. Averaged within each group of models, the gap widens from 26% on small puzzles to 37% on large ones. Notably, both proprietary models solve at least 50% of the large puzzles, whereas none of the open-weight models solves more than 50% even of the small puzzles. Codoku thus sits at neither ceiling nor floor for current models: our puzzles resolve differences between them.

Larger profiles are generally harder. Every model solves fewer large puzzles than small ones, and the decline is monotonic for all models except GPT 5.6 Sol, which performs about equally on medium and large puzzles. Averaged over the five models, the solve rate falls from 56% (small) to 39% (medium) and 30% (large). The decline is steeper for open-weight models, which retain about 30% of their small-profile solve rate on large puzzles, against about 70% for proprietary ones.

Larger profiles demand more resources. Among solved runs, the median request count, wall-clock time, model cost in dollars, and thinking time all increase from small to medium to large for every model (Figure 3). Models trade these resources off differently. GPT 5.6 Sol is the fastest on every profile, taking a median of 10.1 minutes per solved large puzzle against 19.1 for Claude Opus 5, although the former issues nearly twice as many requests. DeepSeek V4.1 Flash is the cheapest on every profile yet issues the most requests, whereas Claude Opus 5 is the most expensive, at a median of $4.77 per solved large puzzle. GLM 5.3 and Kimi K3 are by far the slowest: their median solved large puzzle takes 42.2 and 52.8 minutes, close to the one-hour budget. This suggests that the time limit contributes to their high exhaustion rates on large puzzles; indeed, the thinking-time plot shows that both models spend substantial time thinking. A breakdown is presented in Section A.3.

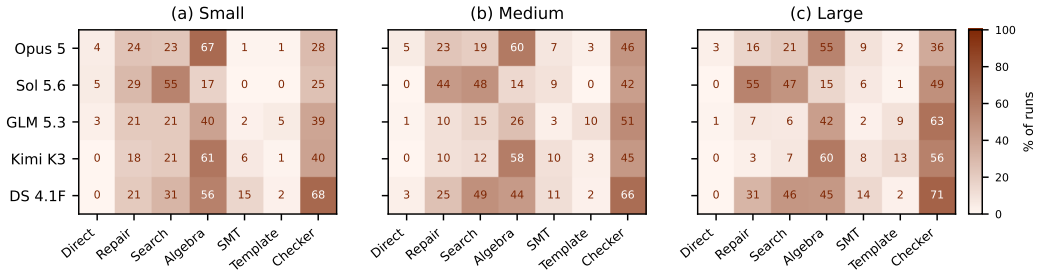


Figure 4: **Agents combine several strategies to solve codokus**. Each cell reports the percentage of solved runs that carry the corresponding strategy label. A run may carry several labels.

Answer to RQ2. Codoku poses a substantial challenge to all five evaluated models under the Pi agent and our budgets: none approaches saturation on any profile, every model solves fewer large puzzles than small ones, and solved runs demand more resources as the profiles grow. The benchmark also separates proprietary from open-weight models, by a margin that widens with puzzle size.

4 Discussion

4.1 Solving Strategies

To characterize how agents solve codokus, we first inspected a sample of trajectories manually and identified seven recurring strategies (see Section A.6 for an example):

- `Direct`: fill all cells in a single attempt;
- `Repair`: revise a candidate after a failed check and check the revision again;
- `Search`: enumerate or sample candidate fillings with an explicit script;
- `Algebra`: infer program states, inputs, or outputs algebraically from the constraints;
- `SMT`: encode local states or fillings as constraints and solve them with an SMT solver;
- `Template`: fill cells through a template of numbered slots and a mapping from slots to fillings;
- `Checker`: inspect the source code of the solution checker.

We then label the trajectories of all solved runs using keyword-based heuristics, counting only high-confidence labels (Figure 4). For instance, a failed check followed by an edit and a recheck counts as evidence of `Repair`, and executing a Python script that uses `Z3` or `CVC5` as evidence of `SMT`.

Agents combine strategies, and the most frequent one differs by model. For every model and profile, a run carries more than one label on average. No strategy is the most frequent for every model: `Algebra` is the most frequent label for Claude Opus 5 and Kimi K3 on every profile, `Checker` for DeepSeek V4.1 Flash on every profile, `Search` or `Repair` for GPT 5.6 Sol, and `Algebra` or `Checker` for GLM 5.3. Every model carries each label on at least one profile, except Kimi K3, which never carries the `Direct` label. The share of runs that inspect the checker's source code is higher on large than on small puzzles for every model. Because the checker evaluates candidates against Φ rather than against the witness (Section 2.3), its source reveals how the constraints are checked rather than the fillings of the witness. Agents typically begin solving a puzzle by reading the checker.

Agents infer program states and confine their search. `Direct` appears in at most 5% of the runs of any model and profile, which suggests that agents treat the cells as interdependent. Instead of filling all cells at once, runs derive cell values from the program states that the constraints entail, either algebraically (`Algebra`, 14%–67% of runs) or with an SMT solver (`SMT`, up to 15%). Where agents search, in the trajectories we inspected, they confine the search to restricted domains, such as the constants in the table or values around a "pivot" inferred from a local program state; exhaustive enumeration is infeasible in any case, given the size of the search space (Section 3.2). When a candidate fails a check, agents also revise it and check it again (`Repair`, 3%–55% of runs). Together, these observations indicate that agents solve codokus by reasoning about program states and narrowing their search accordingly, rather than by exhaustively enumerating fillings.

4.2 Limitations and Threats to Validity

Limitations. Codokus are limited by dead blocks, since not every cell lies on the executed path. A cell in a block that the execution never reaches is constrained only statically, and may therefore admit more fillings and be easier to fill; in Figure 1, for instance, the `ID` cell of the unexecuted block `b2` admits all three variables. Input-output examples whose executions jointly cover every basic block would constrain such cells dynamically as well; we leave their generation to future work.

Threats to validity. Our results depend on the time, cost, and request limits. 784 unsuccessful runs end at a limit while the agent is still at work; a detailed breakdown is presented in Section A.4. We adopt limits in line with standard coding-agent setups for repository-level tasks (Wang et al., 2026b;a), which we consider reasonable for the substantially smaller codoku puzzles. Larger budgets may raise solve rates, but doubling the time (2h), cost ($30), and request (256) budgets for GPT 5.6 Sol with the large profile yields only modest gains: just eight additional cases are solved out of 54. This suggests that codoku puzzles remain challenging even with substantially larger budgets. The strategy analysis of Section 4.1 relies on keyword-based heuristics: although we count only high-confidence labels, the heuristics may miss strategies that leave no lexical trace in a trajectory and mislabel others, and our observations on how agents confine their search rest on manual inspection of a sample of trajectories. Finally, codokus are synthetic programs, whose distribution may differ from that of human-written code. We argue that an agent or model claimed to have the ability to reason about programs, however, should be able to reason about synthetic ones as well.

5 Related Work

Program-reasoning benchmarks commonly study the relation among a program, its inputs, and its execution. Codoku differs from them both in how its tasks are specified and in how they are supplied.

Execution and input prediction. CruxEval (Gu et al., 2024) asks a model to predict the output of a short Python function on an input, or an input that yields a given output; CruxEval-X (Xu et al., 2025) extends it to multiple programming languages. CodeI/O (Li et al., 2025) uses both directions as training tasks. REval (Chen et al., 2025) extends prediction to intermediate states, statement coverage, and execution paths, CoRe (Xie et al., 2025) asks about control and data dependencies, and further studies test prediction under code mutations and input perturbations (Becker et al., 2026; Spiess et al., 2026). These benchmarks remain informative for bare models, but each task comes with a complete program that an agent can execute or instrument instead of reasoning about it (Section 1). In a pilot study, GLM 5.2 (OpenCode) solves all 1,600 CruxEval tasks, each within five minutes.

Program synthesis. Programming by example (PBE) asks a solver to construct a program consistent with given input-output examples. Li & Ellis (2024) evaluate LLMs on classic PBE domains, including text-editing problems from SyGuS (Shi et al., 2022) and PROSE (Microsoft, 2025), and CodeARC (Wei et al., 2025) derives PBE tasks for LLM agents from 1,114 functions in HumanEval, MBPP, and APPS (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021), allowing agents to query the hidden target function on new inputs. Because correctness is defined as agreement with the target on all inputs, a checker can only approximate it: comparison with a reference implementation rejects equivalent alternatives, whereas finite testing may accept incorrect programs. CodeARC checks candidates by differential testing with inputs generated by Pynguin and Mokav (Lukasczyk & Fraser, 2022; Etemadi et al., 2025), yet with the same model and agent under a five-minute limit per task, its success rate ranges from 57.4% to 72.6% depending on the strictness of the checker. Codoku instead makes the specification explicit: correctness is defined by the global constraints of a puzzle alone, so the checker decides it exactly, accepting every filling that satisfies them, including fillings that differ from the witness, and rejecting all others.

6 Conclusion

We presented Codoku, a renewable benchmark that evaluates program reasoning in coding agents through constrained program synthesis. Codoku generates fresh, controlled puzzles with known solutions and explicit semantic constraints. Despite full tool access and compact puzzle sizes, current agents remain challenged, with the best solving 54% of large puzzles. Codoku complements existing benchmarks by providing scalable and automatically checked synthesis tasks for coding agents.

AI USE STATEMENT

In this work, we used generative AI tools to polish the paper's writing, create figures from evaluation data, and assist in writing the puzzle-masking and solution-checking scripts. We did not use these tools to draft sections of the paper, retrieve or discover literature (*e.g.*, related work), develop research ideas, conduct experiments, or generate synthetic datasets. This work does not involve mathematical claims requiring AI-assisted proofs. We reviewed all AI-assisted outputs. This included proofreading the text, verifying that the figures accurately represent the evaluation data, and having two authors test the AI-assisted code for correctness. We take full responsibility for the final content of this work, including all text, claims, and artifacts produced with the assistance of generative AI.

REFERENCES

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

BasLeijdekkers. Metricsreloaded, 2015. URL https://plugins.jetbrains.com/plugin/93-metricsreloaded. Accessed: Sep 8th 2026.

Norman Becker, Tural Mammadov, and Andreas Zeller. Can LLMs really reason about code? studying how well LLMs understand the relation between input, code, and output. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, AIware '26, pp. 21–30, 2026. doi: 10.1145/3805760.3814888. URL https://doi.org/10.1145/3805760.3814888.

Dirk Beyer. Depdigger: A tool for detecting complex low-level dependencies, 2009. URL https://www.sosy-lab.org/%7Edbeyer/DepDigger/. Accessed: Sep 8th 2026.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Li Ge, and Xin Xia. Reasoning runtime behavior of a program with LLM: How far are we? In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE '25, pp. 1869–1881, 2025. doi: 10.1109/ICSE55347.2025.00012. URL https://doi.org/10.1109/ICSE55347.2025.00012.

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

Kavya Chopra, Cong Li, Thodoris Sotiropoulos, and Zhendong Su. Semantic reification: A new paradigm for random program generation. *Proc. ACM Program. Lang.*, (PLDI), 2026. doi: 10.1145/3808268. URL https://doi.org/10.1145/3808268.

Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. Mokav: Execution-driven differential testing with llms. *J. Syst. Softw.*, 2025. doi: 10.1016/j.jss.2025.112571. URL https://doi.org/10.1016/j.jss.2025.112571.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML '24, pp. 16568–16621, 2024. URL https://openreview.net/forum?id=Ffpg52swvg.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL https://openreview.net/forum?id=sD93GOzH3i5.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. CodeI/O: Condensing reasoning patterns via code input-output prediction. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML '25, 2025. URL https://openreview.net/forum?id=feIaF6vYFl.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by LLMs? In *Advances in Neural Information Processing Systems*, volume 37 of *NeurIPS '24*, 2024. URL https://openreview.net/forum?id=xqc8yyhScL.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated Unit Test Generation for Python. In *Proceedings of the 44th International Conference on Software Engineering Companion*, 2022. doi: 10.1145/3510454.3516829.

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976. doi: 10.1109/TSE.1976.233837.

Microsoft. PROSE public benchmark suites, 2025. URL https://github.com/microsoft/prose-benchmarks. Accessed: Sep 8th 2026.

Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, ICSE '21, 2021. doi: 10.1109/ICSE43902.2021.00056.

Pi. Pi, 2026. URL https://pi.dev/. Accessed: Sep 18th 2026.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, ICLR '22, 2022. URL https://openreview.net/forum?id=qhC8mr2LEKq.

Claudio Spiess, Premkumar Devanbu, and Earl T. Barr. How robustly do LLMs understand execution semantics? In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, AIware '26, pp. 288–298, 2026. doi: 10.1145/3805760.3814919. URL https://doi.org/10.1145/3805760.3814919.

Kiss Tamás. Codemetrics, 2016. URL https://marketplace.visualstudio.com/items?itemName=kisstkondoros.vscode-codemetrics. Accessed: Sep 8th 2026.

Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, Kurt Thomas, Yan Shoshitaishvili, Wenbo Guo, Jingxuan He, Thorsten Holz, and Dawn Song. ExploitGym: Can ai agents turn security vulnerabilities into real attacks?, 2026a. URL https://arxiv.org/abs/2605.11086.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI agents' real-world cybersecurity capabilities at scale. In *Proceedings of the 2026 14th International Conference on Learning Representations*, ICLR '26, 2026b. URL https://openreview.net/forum?id=2YvbLQEdYt.

Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. CodeARC: Benchmarking reasoning capabilities of LLM agents for inductive program synthesis. In *Proceedings of the Conference on Language Modeling*, COLM '25, 2025. URL https://openreview.net/forum?id=Q5pVZCrrKr.

Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. CoRe: Benchmarking LLMs' code reasoning capabilities through static analysis tasks. In *Advances in Neural Information Processing Systems Track on Datasets and Benchmarks*, NeurIPS '25 (Datasets and Benchmarks Track), 2025. URL https://openreview.net/forum?id=WJIDorHiuZ.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, ACL '25, 2025. doi: 10.18653/v1/2025.acl-long.1158. URL https://aclanthology.org/2025.acl-long.1158/.

Xuejun Yang, Yang Yang, Jerry Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 283–294, 2011.

Table 2: **Puzzle metrics per profile.** Values are rounded to one decimal place. Each profile reports mean with its standard deviation and median with the first and third quartile.

	Small					Medium					Large				
Metric	*Mean*	*Std.*	*Med.*	*Q1*	*Q3*	*Mean*	*Std.*	*Med.*	*Q1*	*Q3*	*Mean*	*Std.*	*Med.*	*Q1*	*Q3*
search space															
Typed cells	58.5	17.4	58.5	46.0	71.3	134.7	62.4	114.5	89.8	168.0	366.5	162.1	313.0	263.0	444.8
`ID`	11.9	4.4	11.5	9.0	15.0	32.6	18.3	26.0	20.0	41.3	95.1	43.0	81.5	65.0	115.0
`FUNC`	0.0	0.0	0.0	0.0	0.0	0.8	1.1	0.0	0.0	1.0	2.9	2.8	2.0	1.0	4.0
`CONST`	14.7	5.7	14.0	10.0	19.0	32.1	12.7	29.0	23.8	39.0	78.6	33.5	69.5	54.8	95.3
`OP`	31.2	10.8	31.0	22.0	40.3	65.9	31.0	56.0	44.0	80.3	177.7	83.8	153.0	122.8	209.3
`CTRL`	0.4	0.6	0.0	0.0	1.0	1.6	1.1	1.0	1.0	2.0	3.6	3.7	3.0	1.0	4.3
`LABEL`	0.3	1.3	0.0	0.0	0.0	1.7	2.8	0.0	0.0	4.0	8.6	6.2	8.0	4.0	12.0
Constant table entries	13.4	5.1	13.0	10.0	18.0	28.2	11.0	26.0	20.8	33.3	66.9	27.3	60.0	46.0	84.3
Log10 space	72.0	22.7	73.4	55.5	87.0	182.3	89.5	152.6	118.4	226.9	540.0	253.4	451.0	378.8	661.3
static structure															
Lines of code	38.9	6.8	38.0	34.8	41.3	71.9	13.2	69.0	63.8	82.0	133.5	22.9	131.0	118.0	147.8
Distinct operators	22.3	2.5	23.0	21.0	24.0	28.1	3.0	28.0	26.0	31.0	33.2	2.7	33.0	31.0	35.0
Distinct operands	36.0	7.7	35.0	31.0	41.0	72.9	16.5	69.0	60.8	82.0	136.4	37.7	129.0	109.8	152.8
Total operators	140.3	41.1	136.0	115.8	162.0	292.8	99.5	266.5	215.5	351.5	708.5	262.0	656.5	524.3	816.5
Total operands	121.4	45.0	113.5	96.0	134.3	303.4	93.3	283.5	226.8	364.0	703.9	210.3	663.5	581.3	783.0
Halstead diff.	38.6	15.1	34.4	28.0	44.8	58.3	12.9	56.7	48.6	66.3	86.2	14.4	84.4	75.1	96.1
CFG nodes	4.8	0.7	5.0	4.0	5.0	6.6	1.1	7.0	6.0	8.0	9.9	1.3	10.0	9.0	11.0
CFG edges	5.5	1.0	5.0	5.0	6.0	8.9	1.8	9.0	7.8	10.0	14.6	2.2	15.0	13.0	16.0
Cyclo. compl.	2.7	0.7	3.0	2.0	3.0	4.3	1.0	4.0	4.0	5.0	6.7	1.3	7.0	6.0	8.0
Data dep. nodes	21.4	5.7	20.0	18.0	22.0	44.0	10.5	41.5	36.8	51.0	93.0	19.3	91.5	79.0	104.5
Data dep. edges	16.7	9.3	15.0	12.8	16.0	46.3	15.9	44.0	33.8	55.3	111.7	31.7	109.5	90.0	130.3
Data dep. degree	1.5	0.3	1.4	1.3	1.6	2.1	0.3	2.0	1.9	2.3	2.4	0.3	2.4	2.2	2.6
Data dep. max degree	6.7	2.5	6.0	5.0	8.0	10.6	3.5	10.0	8.0	12.0	23.3	6.3	23.0	18.0	28.0
Data dep. density (%)	4.0	1.0	4.0	3.0	4.0	2.0	0.0	2.0	2.0	3.0	1.0	0.0	1.0	1.0	1.0
dynamic behavior															
EP length	5.9	1.6	6.0	5.0	7.0	9.7	3.0	9.0	7.0	11.3	15.3	3.8	15.0	12.0	17.0
Unique blocks	4.7	0.7	5.0	4.0	5.0	5.9	1.1	6.0	5.0	7.0	7.1	1.5	7.0	6.0	8.0
Repeated blocks	1.2	1.3	1.0	0.0	2.0	3.8	2.5	3.0	2.0	5.0	8.2	3.7	7.0	6.0	10.0
Max visits per block	1.5	0.5	2.0	1.0	2.0	2.5	0.6	2.0	2.0	3.0	4.1	1.0	4.0	3.0	5.0
Loops	0.7	0.5	1.0	0.0	1.0	1.1	0.3	1.0	1.0	1.0	1.1	0.2	1.0	1.0	1.0
Loop iterations	1.1	0.9	1.0	0.0	2.0	2.5	1.0	2.0	2.0	3.0	4.1	1.0	4.0	3.0	5.0
input-output examples															
Examples	3.7	0.8	3.5	3.0	4.0	6.3	1.1	6.0	5.0	7.0	9.0	0.8	9.0	8.0	10.0

A Appendix

A.1 Puzzle Complexity

Table 2 reports detailed statistics about the 300 puzzles that we use in our evaluation. It extends Table 1a with more metrics. Following the conclusion in Section 3.2, all metrics except two increase from the small to the medium to the large profile. The two exceptions are the density of the data-dependency graph, which decreases from 0.04 to 0.01, and the number of loops, whose median is one in every profile; the number of loop iterations, in contrast, increases across the profiles.

A.2 Tool Calls

We classify tool calls into six categories: `File` for file inspection and editing, `Validate` for syntax checking and solution verification, `Execute` for candidate execution, `Search` for candidate-search activity, `Debug` for debugging and instrumentation, and `Others` for environment operations and remaining calls. Figure 5 reports the median number of calls in each category per successful run, providing tool-level evidence that complements the strategy analysis in Section 4.1. Median calls categorized as candidate search increase across profiles for every model, while direct-execution calls remain between one and three, and debugging calls remain between zero and three. GPT 5.6 Sol also makes more validation calls as profile scale increases. This pattern is consistent with the increasing prevalence of `Repair` strategy in its successful trajectories. DeepSeek V4.1 Flash makes the most tool calls overall on every profile, with medians increasing from 49 on small puzzles to 78.5 on large ones. The tool-level `Search` category and the run-level `Search` strategy label should not be interpreted interchangeably. The former counts operations categorized as candidate-search activity, whereas the latter requires high-confidence evidence of an explicit enumeration or sampling strategy. Frequent

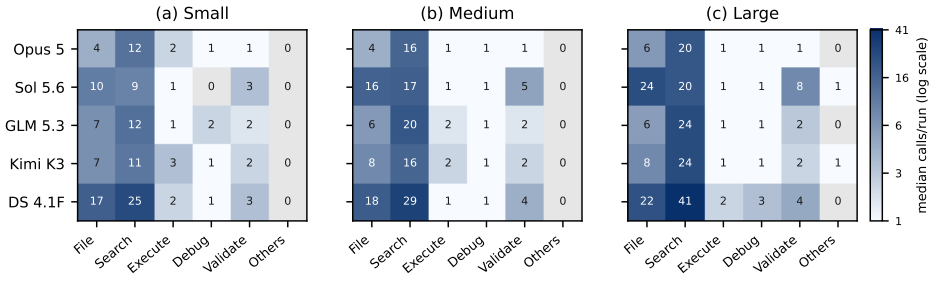


Figure 5: **Agents actively use tools, particularly for candidate search, but codoku puzzles make it difficult to translate tool calls into valid solutions**. `File`: file inspection and editing. `Search`: candidate search. `Execute`: candidate execution. `Debug`: debugging and instrumentation. `Validate`: syntax checking and solution verification. `Others`: all other tool calls.

Table 3: **Average requests, token usage, cost, and thinking time**, reported over all 1,500 agent runs.

			Tokens (K)						
Model	**Profile**	**Requests**	*Input*	*Cache read*	*Cache write*	*Output*	*Think*	**Cost ($)**	**Think (m)**
Claude Opus 5	· *small*	22.8	0.0	1,494.5	105.7	67.5	56.9	3.49	6.6
	· *medium*	29.7	0.1	2,312.4	283.9	97.2	84.6	5.85	9.4
	· *large*	33.6	0.1	2,934.5	368.8	123.2	111.1	7.20	12.3
GPT 5.6 Sol	· *small*	39.7	0.1	1,126.1	39.9	19.9	10.2	1.20	2.2
	· *medium*	60.0	0.6	2,349.9	56.8	24.8	14.6	1.75	3.3
	· *large*	75.1	0.2	3,894.2	72.8	29.1	16.6	2.77	4.0
GLM 5.3	· *small*	21.8	245.1	2,009.6	0.0	124.3	116.5	1.68	14.7
	· *medium*	23.5	307.2	2,478.7	0.0	156.1	149.0	2.20	19.0
	· *large*	28.9	363.9	3,053.8	0.0	169.3	160.7	2.74	21.0
Kimi K3	· *small*	26.8	31.2	1,585.8	113.0	79.1	67.5	2.58	15.2
	· *medium*	29.8	42.1	2,008.4	136.0	96.9	85.8	3.36	19.2
	· *large*	33.1	103.5	2,395.9	151.0	108.7	98.6	4.36	22.8
DPSK V4.1 Flash	· *small*	61.7	60.7	6,911.5	0.0	115.1	99.4	0.22	4.7
	· *medium*	73.9	80.5	9,278.4	0.0	138.4	122.8	0.28	5.9
	· *large*	84.6	97.5	12,179.5	0.0	154.3	136.3	0.32	6.7
All runs		43.0	88.9	3,735.2	88.4	100.3	88.7	2.66	11.1

search-category calls therefore do not imply that the explicit-search label appears in a comparable fraction of runs.

A.3 Cost Breakdown

Table 3 breaks down the requests, tokens, and cost of every run, including solved and unsuccessful ones, and thus complements Figure 3, which covers solved runs only. Our experiment cost around $4,000 in total: ∼$1,650 for Claude Opus 5, ∼$575 for GPT 5.6 Sol, ∼$665 for GLM 5.3, ∼$1,030 for Kimi K3, and ∼$85 for DeepSeek V4.1 Flash. For every model, the mean number of requests, the mean number of tokens, and the mean cost per run increase from small to medium to large puzzles, just as the median requests and cost increase among solved runs in Figure 3.

A.4 Failure Analysis

Termination reasons. Table 4 divides the exhausted and failed runs of Table 1b by the recorded reason for their termination. Of the 784 exhausted runs, 694 reach the time limit, 56 the request limit, and 34 the cost limit. Of the 92 failed runs, 63 end with the agent stopping without a valid solution, 17 with an infrastructure error, 8 with a safety refusal, and 4 with an output-length error. The reasons differ across models. All cost limits, safety refusals, and output-length errors occur in runs of Claude Opus 5, and all request limits in runs of GPT 5.6 Sol and DeepSeek V4.1 Flash.

Table 4: **Termination reason of all agent runs**. The last row sums over all 1,500 runs. *Exhausted* and *Failed* group the reasons as in Table 1b.

			Exhausted			Failed			
Model	**Profile**	**Solved**	*Time*	*Cost*	*Requests*	*Stopped*	*Refusal*	*Length*	*Infra.*
Claude Opus 5	*Small*	77	18	2	0	0	3	0	0
	Medium	62	25	8	0	0	4	0	1
	Large	50	21	24	0	0	1	4	0
GPT 5.6 Sol	*Small*	67	32	0	1	0	0	0	0
	Medium	53	38	0	5	4	0	0	0
	Large	54	24	0	9	12	0	0	1
GLM 5.3	*Small*	50	40	0	0	9	0	0	1
	Medium	22	53	0	0	18	0	0	7
	Large	12	80	0	0	5	0	0	3
Kimi K3	*Small*	48	51	0	0	1	0	0	0
	Medium	28	72	0	0	0	0	0	0
	Large	11	89	0	0	0	0	0	0
DPSK V4.1 Flash	*Small*	39	49	0	12	0	0	0	0
	Medium	29	53	0	12	3	0	0	3
	Large	22	49	0	17	11	0	0	1
All runs		624	694	34	56	63	8	4	17

Table 5: **Fine-grained checker verdict on the final candidate of each run**. *No sol.*: no provided solution. The remaining columns give the first check that the candidate fails. *Basics* checks if all cells are filled, *Re-mask.* is the cell and structure preservation check, and *Exec. limit* is the checker's execution time limit (5s) for the candidate (Section 2.3).

Model	**Profile**	**No sol.**	**Basics**	**Parse**	**Compile**	**Re-mask.**	**CFG**	**Exec. limit**	**EP**	**Output**	**Const.**	**Pass**
Claude Opus 5	· *small*	22	0	0	0	0	0	1	0	0	0	77
	· *medium*	37	1	0	0	0	0	0	0	0	0	62
	· *large*	47	0	0	0	0	0	0	3	0	0	50
GPT 5.6 Sol	· *small*	32	0	0	0	0	0	0	1	0	0	67
	· *medium*	42	0	0	0	2	0	0	0	3	0	53
	· *large*	43	0	0	0	1	0	0	2	0	0	54
GLM 5.3	· *small*	46	0	0	0	3	0	0	1	0	0	50
	· *medium*	74	0	1	0	1	0	0	2	0	0	22
	· *large*	83	0	1	0	3	1	0	0	0	0	12
Kimi K3	· *small*	50	0	0	0	2	0	0	0	0	0	48
	· *medium*	68	0	0	0	4	0	0	0	0	0	28
	· *large*	84	0	0	0	2	0	1	2	0	0	11
DPSK V4.1 Flash	· *small*	59	1	1	0	0	0	0	0	0	0	39
	· *medium*	65	0	0	0	2	0	0	4	0	0	29
	· *large*	63	1	1	0	1	1	0	11	0	0	22
All runs		815	3	4	0	21	2	2	26	3	0	624

Fine-grained checker verdicts. Table 5 reports the solution checker's verdict on all agent runs. Runs that end at a budget can still carry a final candidate solution to check: all 89 unsuccessful runs of Kimi K3 on large puzzles reach the time limit, and 5 of them end with a candidate that fails a check. Of the 876 unsuccessful runs, 815 end without a candidate solution. The remaining 61 end with a candidate that fails a check: 26 at the execution path, 21 at re-masking, and 14 at the other checks combined. No final candidate fails at compilation or at the constant check. Across the runs, every solved run executes a candidate at least once, with a median of five executions per run, whereas 17% of the unsuccessful runs do so, with a median of zero.

A.5 Budgets and Effort Scaling

We evaluate whether codokus remain challenging with larger resource budgets and higher reasoning effort. We use the large profile and select GPT 5.6 Sol because it performs best on this profile (Table 1b), leading to 54 puzzles. Budget constraints prevent us from conducting more experiments.

Double budget. We double the time, cost, and request budgets to 2 hours, $30, and 256 requests. This yields eight additional solved puzzles out of 54, while 40 puzzles still exceed the time limit.

Xhigh effort. We increase the reasoning effort to `xhigh`. This yields 11 additional solved puzzles out of 54, while 23 puzzles still exceed the time limit.

A.6 Case Study

We analyze a small Codoku puzzle and how agents solve it.

Puzzle. The puzzle has five basic blocks (Lines 3–6). Four of them, *i.e.*, `entry`, `b0`, `b1`, and `b2`, form a loop (Lines 51–68), but the execution follows a straight-line path because the loop runs for only one iteration, as specified by the execution path (Line 15). The constant table contains 13 entries with 18 total occurrences, with at most three occurrences per constant (Line 43). The puzzle provides three input-output examples (Lines 73–75) that call `codoku`. Four checksums constrain the filling: the return value of each of the three example calls (Lines 73–75) and the accumulated checksum after the third call (Line 76). This structure resembles sudoku puzzles, which imposes local constraints on each 3×3 block and global constraints across the 9×9 grid. In codoku puzzles, the three return-value checksums act as local constraints, while the accumulated checksum acts as a global constraint across the examples. The banner comment lists additional requirements that a valid solution must satisfy (Lines 29–32). These features are consistent with the design described in Section 2.

Solution. For this small puzzle, Claude Opus 5 solved it in ∼36 minutes while GPT 5.6 Sol, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash all timed out with a one-hour budget. Figure 7 displays Claude's solution.

Overview. Claude solves the puzzle by inspecting the checker, deriving checksum constraints, and combining enumeration with randomized search. It then fills the remaining cells to satisfy the required control flow and constant table, assembles the solution, and passes the checker on its first attempt.

Phase 1: read the puzzle. From the puzzle, Claude infers that the loop runs once per call and the branch in `b2` (Lines 64–65) is not taken. Because `v2` and `t0[2]` are fixed at Lines 67–68 before the `exit` block, the agent identifies `v0`, `v1`, and `t0[0]` as unknown exit values and decides to inspect the checker before searching.

Phase 2: inspect the checker. The agent reads the checker and its helper module to establish the filling rules. Unary operators are allowed before constants, but `and` and `or` cannot fill `OP` cells; parameters and subscripts can fill `ID` cells; and the 18 `CONST` cells must reproduce the constant table exactly.

Phase 3: reason local checksums. The agent simplifies the exit checksums using the known inputs inferred before and some shifts inferred by reading the `global_chksum` function. It reduces unknowns to `v1`, `t0[0]`, and the sign of `v0`. Enumerating the `b1` and `b2` blocks with templates yields 824 matching combinations, all with `v1 = 67108870 + (2 * pa0)` and `t0[0]` equal to `0` in every call. It later makes the latter happen using `t0[0] = pa2 - pa2`.

Phase 4: reason global checksum. Based on prior reasoning, the agent expresses the accumulated checksum as

$$\sum_{i=1}^{3} \left((y_i + A) \bmod m_i\right) - \sum_{i=1}^{3} K_i,$$

where mod is Python's remainder, y_i is the intermediate value after the second shift in `global_chksum`, $A = 7328\,c_3 - 1651\,c_4$ depends on the initial values, and m_i and K_i depend on each call's inputs. Using $y_1, y_3 \in \{0, -1\}$, it reduces the checksum to $y_2 \equiv \rho \pmod{944{,}660{,}365}$. It leverages the GNU `factor` utility to factor the candidate numbers. This produces 17 candidate values for `v1`, but these appear difficult to realize with the `b0` template, so it turns to randomized search.

Phase 5: perform randomized search. The agent samples fillings of the `b0` template and filters them using modular constraints. After *96 million* unsuccessful samples, it verifies its algebra and finds that the search produces too few distinct values. Restricting the operators improves diversity, and it yields a match within the allowed time budget. The match fixes the `b0` expression and the initial values: `t0[0]` $= -62$ and `v1` $= -95$.

Phase 6: fill remaining cells. Claude then fills the control-flow cells so that the loop runs once, the `b2` branch is not taken, and the `CTRL` cell preserves the declared CFG (Lines 3–6). It chooses assignments

```
# codoku() is a function of the following CFG:
#
#@CFG_EDGE: entry -> b0, exit
#@CFG_EDGE: b0 -> b1
#@CFG_EDGE: b1 -> b2
#@CFG_EDGE: b2 -> entry, exit
#
# ------------------------------------------------
# Task
# ------------------------------------------------
#
# Replace all occurrences of <XXX> with appropriate code to make the function return the expected values for the examples
# in main following the below execution path:
#
#@EXE_PATH: entry -> b0 -> b1 -> b2 -> entry -> exit
#
# ------------------------------------------------
# Validation
# ------------------------------------------------
#
# Use the following command to verify your solution:
#
#   codoku check puzzle.py solution.py
#
# ------------------------------------------------
# General Requirements
# ------------------------------------------------
#
# 1. Each <XXX> cell must be filled out with a corresponding element.
# 2. You have access to all common command line tools and SMT solvers.
# 3. Do NOT change any code except for the <XXX> cells.
# 4. Do NOT introduce any new code, variables, or basic blocks.
#
# ------------------------------------------------
# Requirements for <CONST>
# ------------------------------------------------
#
# The line below list every constant the <CONST> cells must carry, as '<value>:<count>' pairs. Across your whole solution
# each <value> must appear in <CONST> positions exactly <count> times -- no more, no fewer -- and no other constant may
# appear in any <CONST> position. The value must match exactly, including its type: `2` (integer) and `2.0` (float) are
# distinct. Constants already shown in the fixed code do not count toward this budget.
#
#@CONST_TB: 1048577:1, 1946157169:3, 2:2, 2231:1, 24:1, 251:1, 32:1, 51:1, 62:1, 67108870:1, 71:1, 74261:1, 805306406:3
g__chk = [0]

def codoku(pa0, pa1, pa2):
  v0 = OP CONST;  v1 = OP95;  v2 = -CONST;  t0 = [-CONST, _PAD, OP CONST, _PAD, _PAD];
  #@CFG_BLOCK entry
  t0[2] = ((cast_int(-3, 20) + cast_int(v1, 20)) - -524288)
  v2 = (cast_int(1912602622, 64) - (-13 * pa1))
  while (((1 OP ID) OP (OP1946157169 OP pa0 OP (~CONST OP pa0 OP (OP CONST OP 0)
          == (ID OP 0) OP -(OP OP CONST OP ID))))) OP ((OP CONST OP ID)):
    #@CFG_BLOCK b0
    v1 = ((CONST - ID OP (805306406 OP ID OP (CONST OP 0) OP (ID < 0) OP -(OP CONST OP pa0)))
           - (OP CONST OP ID))
    v0 = (cast_int(CONST, 8) OP cast_int(v1, 8))
    #@CFG_BLOCK b1
    g__chk[0] = cast_int(global_chksum(g__chk[0], cast_int(v0, 32), v1, cast_int(v2, 32), ...), 64)
    v0 = (cast_int(CONST, 8) OP cast_int(ID, 8))
    t0[0] = (pa2 OP ID)
    #@CFG_BLOCK b2
    v2 = ((cast_int(OP CONST, 64) OP OP2147483647) - ID)
    v1 = (CONST OP (CONST * pa0))
    if ((ID OP (OP32746 OP ID))) OP (((cast_int(OP1, 16) OP 0) OP (0 OP (ID) OP (0) OP CONST))):
      CTRL
    #@CFG_BLOCK entry
    t0[2] = ((cast_int(-3, 20) + cast_int(v1, 20)) - -524288)
    v2 = (cast_int(1912602622, 64) - (-13 * pa1))
  #@CFG_BLOCK exit
  return global_chksum(0, cast_int(v0, 32), v1, cast_int(v2, 32), cast_int(_rd(t0, 0), 32), ...)

if __name__ == '__main__':
  r = codoku(-33554434, -4294967296, 22); check_chksum(-865773859, r)
  r = codoku(-546995, 576460752399126213, 11512); check_chksum(-667196848750, r)
  r = codoku(-131331, 0, -15130); check_chksum(-84085712, r)
  check_chksum(-667895151815, g__chk[0])
```

Figure 6: **A small codoku puzzle example**. Claude Opus 5 solved it in ∼36 minutes while GPT 5.6 Sol, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash timed out with a one-hour budget.

to `v0` that satisfy the checksum sign constraints (according to Phase 3) and distributes the remaining constants among checksum-irrelevant cells to match the table.

```
def codoku(pa0, pa1, pa2):
  v0 = - 1946157169;  v1 = -95;  v2 = -1946157169;  t0 = [-62, _PAD, - 1946157169, _PAD, _PAD];
  #@CFG_BLOCK entry
  t0[2] = ((cast_int(-3, 20) + cast_int(v1, 20)) - -524288)
  v2 = (cast_int(1912602622, 64) - (-13 * pa1))
  while (((1 + t0[0]) > (-1946157169 * pa0 * (-2231 * pa0 - (- 24 * 0)
          == (v0 * 0) * -(- - 51 * v1))))) < ((- 805306406 < pa2)):
    #@CFG_BLOCK b0
    v1 = ((805306406 - pa0 * (805306406 // pa0 - (32 - 0) + (pa0 < 0) % -(+ 1048577 // pa0)))
           - (not 74261 // pa2))
    v0 = (cast_int(2, 8) + cast_int(v1, 8))
    #@CFG_BLOCK b1
    g__chk[0] = cast_int(global_chksum(g__chk[0], cast_int(v0, 32), v1, cast_int(v2, 32), ...), 64)
    v0 = (cast_int(251, 8) + cast_int(t0[0], 8))
    t0[0] = (pa2 - pa2)
    #@CFG_BLOCK b2
    v2 = ((cast_int(- 71, 64) * -2147483647) - t0[0])
    v1 = (67108870 + (2 * pa0))
    if ((t0[0] + (-32746 * pa2))) * (((cast_int(-1, 16) * 0) * (0 + (t0[0]) + (0) + 805306406))):
      break
    #@CFG_BLOCK entry
    t0[2] = ((cast_int(-3, 20) + cast_int(v1, 20)) - -524288)
    v2 = (cast_int(1912602622, 64) - (-13 * pa1))
  #@CFG_BLOCK exit
  return global_chksum(0, cast_int(v0, 32), v1, cast_int(v2, 32), cast_int(_rd(t0, 0), 32), ...)
```

Figure 7: **Filling submitted by Claude Opus 5** for the puzzle of Figure 6. Highlighted tokens fill the cells; the remaining code is fixed. The filling is valid.

Phase 7: assemble and check. The agent assembles the 80 possible fillings with a Python script and executes them to confirm the required execution path for all three calls. Then it checks them through the solution checker, which returns `PASS` for two fillings. Claude submits one of them.

Observations. Consistent with Section 4.1, the agent inspects the checker source (`Checker`), algebraically derives constraints on intermediate values (`Algebra`), searches for fillings through scripted enumeration and sampling (`Search`), and fills the cells using a numbered list of slots (`Template`). It does not use an SMT solver in this run, but invokes GNU `factor` to compute prime factorizations. The run does not include `Repair`, as two of the 80 initial fillings already pass solution verification. Phases 4 and 5 consume 25.8 of the run's 36.4 minutes and $3.70 of its $6.07 cost. Some fillings reduce subexpressions to constants, as in `t0[0] = pa2 - pa2` and the zero factor in the `b2` condition; these simplifications help Claude find a solution more quickly. The checker accepts these fillings because it verifies the global constraints rather than requiring agreement with the witness (Section 2.3).