Supplementary Information

Supplementary Note 1: Benchmark

Supplementary Table 1: Overview of the SemBench

Problem counts by category		
Dead code - statement		2 000
Data dependency		1 996
Function reachability		3 548
Dominators		1 965
Dead code - loop		3 930
Liveness		1 965
Feature statistics		
File size (bytes)	3 108	(min: 85, max: 491 196)
Max nesting depth	2	(min: 0, max: 10)
Avg dependency distance	21	(min: 0, max: 630)
Max dependency distance	67	(min: 0, max: 2 071)

Supplementary Table 2: Shorthand mapping in other tables and full model identifiers.

Shorthand in other tables	**Full repo/API identifier**
GPT-5	`openai/gpt-5` (openai.responses.create API)
GPT-5-Codex	`openai/gpt-5-codex` (openai.responses.create API)
GPT-4o Mini	`openai/gpt-4o-mini` (openai.responses.create API)
GPT-3.5 Turbo	`openai/gpt-3.5-turbo` (openai.responses.create API)
DeepSeek-Coder V2-Lite-Instr	`deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct`
DeepSeek-Coder 7B-Instr v1.5	`deepseek-ai/deepseek-coder-7b-instruct-v1.5`
DeepSeek-R1-Distill-Qwen-7B	`deepseek-ai/DeepSeek-R1-Distill-Qwen-7B`
CodeLlama-13B-Instr	`codellama/CodeLlama-13b-Instruct-hf`
CodeLlama-7B-Instr	`codellama/CodeLlama-7b-Instruct-hf`
Llama-3 8B-Instr	`meta-llama/Llama-3.1-8B-Instruct`
Mistral 7B-Instr (v0.3)	`mistralai/Mistral-7B-Instruct-v0.3`
Mamba Codestral 7B (v0.1)	`mistralai/Mamba-Codestral-7B-v0.1`
Qwen2.5-Coder 14B-Instr	`Qwen/Qwen2.5-Coder-14B-Instruct`
Qwen3 14B	`Qwen/Qwen3-14B`
StarCoder 2 7B	`bigcode/starcoder2-7b`
Phi-4 Reasoning (14B)	`microsoft/Phi-4-reasoning`

SemBench Statistics

Supplementary Table 1 gives an overview of key attributes of SemBench, including the count of questions in each semantic category, and attributes of the files.

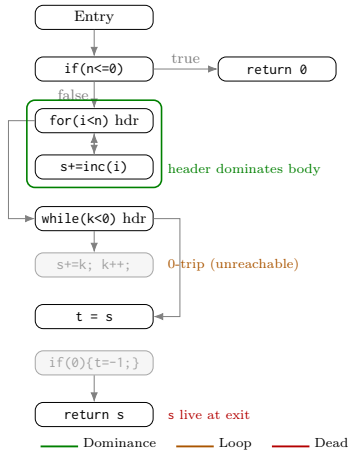
Notations in Tables

For brevity, we use abbreviated model names in the tables of the main manuscript. Supplementary Table 2 maps these shorthand labels to the corresponding full repository or API identifiers.

Ground Truth Collection Methods

```c
int inc(int a){ return a+1; }
int sum_pos(int n){
  int s = 0;
  if (n <= 0) return 0;
  // dominates early exit
  for (int i=0;i<n;i++) {s+=inc(i);}
  // may run if n>0
  int k = 0;
  while (k < 0) { s+=k; k++; }
  // 0-trip => unreachable body
  int t = s;
  // def(t), no later use
  if (0) { t = -1; } // dead branch
  return s; // s live at exit
}
int main(){
   int r=sum_pos(3);
   printf("%d\n",r);
}
```

(a) Running example.



(b) Illustrative graphs extracted.

Supplementary Figure 1: Compact illustration of signals used by SemBench-QE: (a) code example; (b) extracted data and control flow graphs.

Dead Code - statement Dead code detection queries determine the executability of program statements. Our pipeline employs Clang's Static Analyzer, specifically its interprocedural, path-sensitive dead-code checker [1]. Statements flagged by the analyzer as unreachable under any execution path are marked as dead. The Static Analyzer's rigorous and extensively validated techniques [2] guarantee high reliability for these ground-truth labels.

Data Dependency. Data dependency queries examine whether the value of a destination variable a depends on a source variable b. To obtain ground truth, we perform AST-based token analysis using Clang's libclang AST interface [3]. Given a C file, we first parse its AST and collect a set of atomic statements per function. For each collected statement, we extract the defs (i) *defs*, i.e., the variable being defined, and (ii) *uses*, i.e., the set of variables referenced in the initializer or right-hand

side expression, obtained by traversing only within the statement cursor's expression subtree. We then compute dependencies by a single forward pass over the statements in source-line order. We maintain a map from variable names to their most recent defining statement within the same function. When a statement defines a and uses a variable b that has a previous definition in the function, we add a dependency edge $b \rightarrow a$ from the last defining statement of b to the current statement defining a. Finally, we instantiate binary queries over (a, b) pairs from the extracted edges, yielding function-scoped ground truth labels for data dependency. main-text Figure 5(c) illustrates how extracted $(\texttt{fun}, a, b)$ tuples are converted into diverse natural-language questions.

Function Reachability Function reachability queries explore whether a function can potentially invoke another within a call graph derived from static analysis. The pipeline is able to utilize tools such as Clang's Abstract Syntax Tree (AST) parsing capabilities [4] to accurately identify function declarations and calls. A directed call graph is constructed where nodes represent functions and edges denote explicit call relationships. Reachability checks (e.g., depth-first or breadth-first search) provide precise ground-truth, as C's static call semantics exclude indirect calls unless explicitly encoded via function pointers [5]. Consequently, ground-truth answers for function reachability are exact, reflecting definitive interprocedural call paths. As shown in Supplementary Figure 1, the question template is "Is it impossible for function $fun1$ to reach function $fun2$?" The call graph connectivity provides us with ground-truth, and we save the $fun1$, $fun2$, and A as the semantic information to fill into the template.

Dominators At the code level, dominance captures mandatory execution order between program blocks. The block A is said to dominate the block B if every execution of B must first pass through A. Specifically, we compile each C file to LLVM IR with debug information enabled (e.g., `-g` and `-O0`) and parse the emitted IR into a per-function CFG whose nodes are IR basic blocks and whose edges are derived from terminator instructions (e.g., conditional and unconditional branches and switches). We associate source lines with IR blocks using instruction-level debug metadata (`!dbg` tags that reference `DILocation` records), producing a line-to-block mapping. We then compute each function's dominator sets by the standard fixed-point formulation: $\mathrm{Dom}(\mathrm{entry}) = \{\mathrm{entry}\}$ and $\mathrm{Dom}(n) = \{n\} \cup \bigcap_{p \in \mathrm{Pred}(n)} \mathrm{Dom}(p)$ for other nodes [6]. Finally, we lift block-level dominance to statement-level dominance by mapping each statement to its associated IR block (via its source line), and labeling a pair (a, b) as positive if the block of statement a dominates the block of statement b (strict dominance excludes the trivial $a = b$ case). For reference, LLVM also exposes dominator tree analyses for IR functions [7]; our implementation follows the same dominance definition.

Dead Code - loop Dead Code - loop determines whether loops can potentially be skipped or are guaranteed to be executed. We employ LLVM's Scalar Evolution (SCEV) analysis [8] to identify loops with determinable trip counts. If the maximum iteration count computed by SCEV is zero, loops are marked unreachable; otherwise, loops are conservatively considered reachable. The correctness of Scalar–Evolution analysis—widely used in compiler optimisations—is well established [3], rendering our ground-truth labels robust.

Liveness Variable liveness analysis pinpoints variables still holding meaningful values ("live") at the exit points of functions. Our pipeline approximates standard data-flow analysis [6] by first identifying the functions and variables, then using clang CFG to collect live variables at the block exit. Finally, we collect variables used in the return statement, which aligns with the liveness analysis [3], ensuring a reliable ground truth.

Task Design Mindmap To make SemBench a robust and decomposable benchmark, we intentionally design task categories with varying levels of difficulty. Dead Code - statement relatively simple properties, as they can be determined from localized control-flow information. In contrast, Liveness requires reasoning over global program structure and long-range interactions, making them substantially more challenging.

This stratified design enables controlled analysis of different semantic capabilities and their combinations within a unified benchmark. While static analysis cannot capture all aspects of program semantics in real-world programs, SemBench prioritizes correctness over coverage: every semantic property included in the benchmark is derived from sound, deterministic analyses.

Evaluation Pipeline

As seen in main-text Figure 5(c), we also built the automated evaluation pipeline. Same as discussed in section 3, we concatenate four components to form the rich-context initial input from the constructed question bank. Then we feed the initial input into the LLMs to get $\hat{A}$. Afterwards, we will pass $\hat{A}$ through our well-designed output parser to extract the LLMs' answer.

The output parser is designed to reduce mislabeling outputs produced by LLMs. To guarantee the correctness of the evaluation, parsing $\hat{A}$ correctly matters. Our output parser processes $\hat{A}$ by following these steps:

- Cleaning non-English characters and non-instructed symbols.
- Search for and collect declared patterns in the Ins within each line.
- Remove ambiguous answers, e.g., yes or no when it does not clearly answer.

Besides syntactically verifying its correctness, we also tested the parser over sixty real outputs generated by Phi-4-reasoning, which tends to produce diverse, noisy outputs. And our output parser passes all the tests.

Overall, our automated analysis pipeline combines rigorously validated compiler analyses [1, 7–10] and established compiler theory [3, 5, 6] to produce a semantically accurate, robust benchmark dataset suitable for evaluating code-semantic understanding by Large Language Models (LLMs).

Preliminary SemBench-Python Statistics

Supplementary Table 3 summarizes the current SemBench-Python subset derived from CodeNet programs. The benchmark contains 200 Python files with generated semantic

Supplementary Table 3: Overview of the SemBench-Python

Problem counts by category		
Dead code - statement		780
Data dependency		796
Function reachability		734
Dominators		800
Dead code - loop		390
Liveness		780
Feature statistics		
File size (bytes)	2 148	(min: 312, max: 17 133)
Lines of code	83	(min: 12, max: 3 105)
Function count	7	(min: 2, max: 29)
Max nesting depth	4	(min: 1, max: 13)
Avg dependency distance	10	(min: 2, max: 41)
Max dependency distance	73	(min: 2, max: 3 002)

questions. File-level statistics are computed directly from the source programs. Dependency distance is measured as the absolute source-line distance between the queried variable definition and dependency source in the extracted dependency records.

Complementary Literature Review

The introduction has discussed functional-correctness benchmarks, execution-oriented semantic benchmarks, and earlier studies on static reasoning. Here, we further compare SemBench with CoRe [11], the closest concurrent benchmark to our work.

CoRe and SemBench share the goal of moving beyond functional correctness and directly evaluating LLMs' reasoning over static program properties. Their main overlap lies in dependency-oriented task design. CoRe's data-dependency task is closely related to the data-dependency category in SemBench, and its control-dependency task is related to SemBench categories that require control-structure reasoning, such as dominators and dead-code/loop reachability. However, SemBench differs from CoRe in research objective, semantic coverage, label construction, program distribution, analysis granularity, and model coverage.

First, the two benchmarks address different research questions. CoRe evaluates whether LLMs can perform dependency-centered static reasoning, asking models to identify dependency relations, generate dependency traces, or enumerate dependency sources within selected target functions. SemBench additionally asks whether static semantic understanding is related to code-generation capability. To answer this question, SemBench renders six static semantic properties as deterministic yes/no questions and correlates model accuracy with HumanEval and MBPP pass@1 scores. Our results show that this relationship is selective: overall SemBench accuracy only weakly aligns with code-generation performance, whereas function reachability shows the strongest correlation with both HumanEval and MBPP. Thus, SemBench not only evaluates

static semantic understanding, but also identifies which semantic capabilities are most predictive of coding performance.

Second, the task scope is different. CoRe focuses on data dependency, control dependency, and information flow, which are important forms of dependency reasoning at variable and line/statement levels. SemBench covers six static semantic properties: dead-code statements, data dependency, function reachability, dominators, dead-code loops, and liveness. This design extends the evaluation beyond dependency reasoning to include dead-code detection, dominance, liveness, and inter-procedural function reachability. It also enables fine-grained category-level analysis. For example, although dead-code statements and dead-code loops both concern reachability, they exhibit different performance patterns across models.

Third, the label construction methodology is different. CoRe uses a semi-automated annotation pipeline with substantial human verification, which supports high-quality multilingual dependency labels. SemBench instead deliberately selects semantic properties whose questions and ground-truth labels can be generated deterministically from AST, LLVM, and compiler analyses. This design makes the benchmark scalable and reproducible, while avoiding manual semantic annotation for each program instance.

Fourth, the program distribution is different. CoRe contains 12,553 task instances from 180 annotated target functions, with one target function selected from each file and function lengths balanced between 21 and 100 lines of code. SemBench evaluates 1,000 real-world C programs selected from a large CodeParrot-derived pool, with program lengths ranging from 3 to 3,756 lines of code. Each SemBench program contains at least one function and may provide whole-program context for static semantic questions. Therefore, even for overlapping categories such as data dependency, the two benchmarks evaluate models under different program distributions and context scales.

Fifth, the analysis granularity differs. CoRe explicitly focuses on intra-procedural dependency analysis and excludes inter-procedural reasoning. SemBench includes function reachability, which is an inter-procedural property requiring reasoning over the static call graph. This distinction is important because whole-program semantic understanding requires models to track relationships beyond a single target function.

Sixth, the model coverage is complementary. Whereas CoRe emphasizes commercial models and large open-weight models, SemBench evaluates a broader spectrum that includes both proprietary GPT-series models and accessible 7B–14B open-source models. This coverage allows us to study not only peak performance, but also how static semantic understanding varies with model scale, family, instruction tuning, and model evolution. Such coverage is important because smaller open models are reproducible, inspectable, and widely used by the research community.

Finally, to examine whether the observed trends are specific to C, we further construct a preliminary SemBench-Python subset using analogous semantic categories. The results show positive cross-language correlations between SemBench and SemBench-Python, especially for DeadCode-S, FuncReach, and DeadCode-L. We also compare SemBench-Python with MBPP, where a larger matched subset is available than HumanEval in the current open-source-only evaluation. The strongest

correlations appear in FuncReach and DeadCode-S, consistent with the main SemBench finding that the relationship between static semantic understanding and code-generation ability is selective rather than uniform across semantic categories.

Supplementary Note 2: Prompt Library

Full Prompts

As described in Section 3, each prompt to the LLM is constructed by concatenating four parts:

$$(Ins,\ B,\ t_{\text{code}},\ Q),$$

where

- Ins is the *instruction* to the model to format the output;
- B is the *background knowledge* describing the category of each question;
- t_{code} is the source C code;
- Q is the specific question text (generated via the templates).

Below we give the full text of the Ins and B.

Instruction (Ins):

> Given the C code and background information, your response should start with either: `[Final answer: yes]` or `[Final answer: no]`, and then briefly explain your reasoning step by step.

Background knowledge (B) for each semantic category: We supply the model with relevant definitions for each task. The background prompts are as follows:

Dead Code - Statement	Identifies segments that are never executed, such as code following an unconditional return or unreachable branches.
Data Dependency	Variable x depends on variable y if the value of x is determined by the value of y.
Function Reachability	Examines whether one function can transitively call another (i.e., if a call path exists).
Dominators	Statement x dominates statement y if every path passes statement y must go through statement x.
Dead Code - Loop	Examines whether a loop is executed during the program execution based on its condition and structure.
Liveness	A variable x is live at some point if it holds a value that may be needed in the future.

Question template (Q) for each semantic category:

Dead Code - Statement	Is the statement {`code`} unreachable during execution? Can the statement {`code`} be considered dead code because it is never executed?
Data Dependency	Does the value of {`var_a`} depend on the value of {`var_b`}? Does {`var_b`} determine the value stored in {`var_a`}?

Function Reachability	Is there a call path from function {`fun1`} to function {`fun2`}? Can function {`fun1`} eventually call function {`dst`}?
Dominators	Does the statement {`a`} dominate {`b`}? Is it guaranteed that whenever {`b`} is reached, the statement {`a`} has already been reached?
Dead Code - Loop	Can the loop with condition {`cond`} be skipped entirely? Is it possible that the loop with condition {`cond`} is never executed?
Liveness	Is variable {`var`} live at the end of function {`func`}? Does variable {`var`} remain in use at the end of function {`func`}?

Concrete Example

```
Given the C code and background information, your response should start with
    an option between "[Final answer: yes]" or "[Final answer: no]" then you
    should explain your solution step by step briefly.

Function_reachability examines whether one function can transitively call
    another (i.e., if a call path exists).

See the C Code and answer the question:
#include <stdio.h>
struct sss{
  int i1:29;
  int i2:4;
  int i3:10;
};
static union u{
  struct sss sss;
  unsigned char a[sizeof (struct sss)];
} u;
int main (void) {
  int i;
  for (i = 0; i < sizeof (struct sss); i++)
    u.a[i] = 0;
  u.sss.i1 = 536870911.0;
  for (i = 0; i < sizeof (struct sss); i++)
    printf ("%x ", u.a[i]);
  printf ("\n");
  u.sss.i2 = 15.0;
  for (i = 0; i < sizeof (struct sss); i++)
    printf ("%x ", u.a[i]);
  printf ("\n");
  u.sss.i3 = 1023.0;
  for (i = 0; i < sizeof (struct sss); i++)
    printf ("%x ", u.a[i]);
  printf ("\n");
```

```
  return 0;
}

Test Question: Is it possible for function 'main' to reach function 'printf'?
```

Ablation Study of *Ins*

To ensure that SemBench performance is not an artifact of prompting, we conduct a detailed ablation comparing two instruction styles: a Chain-of-Thought–requested instruction (Ins_{CoT}) and a Chain-of-Thought–not-requested instruction (Ins_{NoCoT}). These prompts differ fundamentally in whether they require the model to produce explicit intermediate reasoning before the final answer.
Prompt Definitions. We use the following two instruction prompts exactly:

Ins_{CoT} **(CoT requested)**
Given the C code, background information, and test question, your response MUST begin with step-by-step reasoning. Analyze the control-flow, data-flow, and the specific semantic property in detail. Do not skip any intermediate logic. After giving the reasoning, you should give an conclusion between [Final answer: yes] or [Final answer: no].

Ins_{NoCoT} **(CoT not-requested)**
Given the C code, background information, and test question, answer the following question as concisely as possible. Do NOT show reasoning, do NOT explain, and do NOT think step by step. You should give an conclusion between: [Final answer: yes] or [Final answer: no].

Supplementary Table 4: **Accuracy of GPT-5 under Chain-of-Thought vs. No-CoT Instructions.** All values are pass@1 accuracy (%). Differences across all six semantic categories fall within statistical noise, indicating that prompting style does not meaningfully affect semantic reasoning performance.

Category	**Ins**$_{\text{CoT}}$	**Ins**$_{\text{NoCoT}}$
Dead Code - statement	60.00	65.00
Data Dependency	80.00	80.00
Function Reachability	46.43	50.00
Dominators	88.89	88.89
Dead Code - loop	58.33	55.56
Liveness	84.21	84.21
Overall	67.82	69.02

Accuracy Comparison. We selected 141 problems from the most difficult programs, and the result is shown in the Supplementary Table 4. Changing the instruction from

an explicit Chain-of-Thought style to a concise, answer-first style has no meaningful impact on semantic accuracy. The overall gap between Ins_{CoT} and Ins_{NoCoT} remains below one percentage point, and all per-category fluctuations are small and mixed in sign, consistent with evaluation noise rather than systematic gains. This pattern indicates that prompting for step-by-step reasoning does not unlock additional semantic competence; instead, SemBench primarily probes the model's intrinsic structural understanding of code, which appears insensitive to superficial changes in prompting style.

Conclusion. These ablations collectively show that (1) semantic accuracy on SemBench is not driven by CoT prompting, (2) forcing long-form reasoning does not improve structural code understanding, and (3) the concise Ins_{NoCoT} instruction is both more efficient and equally accurate. This confirms that SemBench measures inherent semantic understanding rather than prompting-induced reasoning heuristics.

Supplementary Note 3: Experiments

Supplementary Table 5: Overall and per-category accuracy (%) on SemBench

Model	All	Dead Code-S	Data-Dep	Func-Reach	Dominators	Dead Code-L	Liveness
GPT-5-Codex	80.42	89.05	87.12	97.83	85.22	64.96	64.38
GPT-5	77.56	91.15	87.56	76.44	90.63	64.53	61.01
GPT-4o Mini	69.55	86.40	83.25	58.85	69.99	67.56	56.54
GPT-3.5 Turbo	65.17	58.27	68.69	69.05	56.28	84.50	58.27
DeepSeek-Coder V2-Lite-Instr	62.22	57.40	70.27	77.79	50.10	68.17	54.15
DeepSeek-Coder 7B-Instr v1.5	43.53	51.05	44.30	41.18	44.64	50.41	32.47
DeepSeek-R1-Distill-Qwen-7B	13.99	15.90	7.80	14.40	9.17	18.98	24.07
CodeLlama-13B-Instr	45.68	50.90	48.28	45.38	49.80	38.19	42.85
CodeLlama-7B-Instr	47.92	49.35	46.81	52.31	49.40	47.35	42.85
Llama-3 8B-Instr	48.52	63.40	62.28	48.56	52.25	38.45	33.89
Mistral 7B-Instr (v0.3)	64.17	60.60	74.30	70.12	56.51	68.07	57.51
Mamba Codestral 7B (v0.1)	39.33	30.85	32.57	54.82	38.83	36.54	47.38
Qwen2.5-Coder 14B-Instr	41.22	89.95	27.61	49.38	20.64	46.77	41.42
Qwen3 14B	48.48	61.00	54.56	58.40	51.90	24.89	51.70
StarCoder 2 7B	42.83	61.85	47.08	36.08	41.78	32.49	43.26
Phi-4 Reasoning (14B)	33.22	51.55	65.03	23.73	48.90	18.45	18.73

Experiment Setup

Model selection We evaluate SemBench on 16 popular state-of-the-art LLMs across 7 different families to cover a wide range of architectures. We lean to LLMs trained on code, but also cover reasoning and general models for comparison. Given the Q&A nature of SemBench, we also include many instructed-tuned versions.

Experiment Setting We tested all open-source language models with the HuggingFace Transformers text-generation API (v 4.41) and queried proprietary GPT

models via the OpenAI Python SDK (openai v 1.15). Experiments ran on a SLURM-managed node equipped with four AMD Instinct MI210 GPUs under the ROCm 6.2.4 stack. Exact hyperparameters and launch commands are provided in the scripts.

Evaluation Metric Formally, let a_i denote the pass@1 accuracy (in percentage) of a model on the i-th semantic task, and let K be the total number of semantic task categories. The overall geometric accuracy is defined as:

$$\text{All\_geo\_p1} = \left(\prod_{i=1}^{K} a_i \right)^{\frac{1}{K}}.$$

In our experiments, $K = 6$.

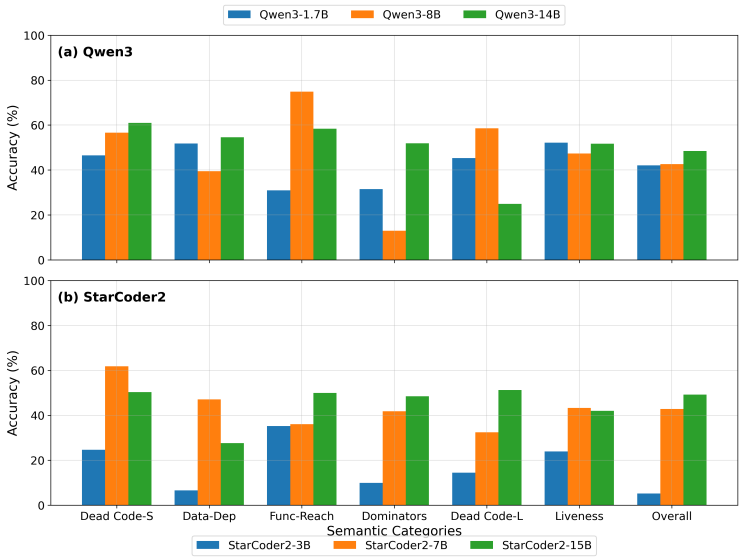
Supplementary Table 6: Overall and per-category accuracy (%) on SemBench-Python

Model	All	Dead Code-S	Data-Dep	Func-Reach	Domin-ators	Dead Code-L	Live-ness
DeepSeek-Coder V2-Lite-Instr	58.62	68.33	68.97	53.13	54.62	61.54	48.21
DeepSeek-Coder 7B-Instr v1.5	47.50	45.13	50.13	50.41	52.12	49.74	38.85
DeepSeek-R1-Distill-Qwen-7B	12.66	17.18	10.55	11.31	12.75	10.77	14.62
CodeLlama-13B-Instr	49.35	39.62	57.79	48.23	51.00	52.82	48.59
CodeLlama-7B-Instr	50.62	49.36	55.90	49.18	50.88	50.26	48.46
Llama-3.1 8B-Instr	35.89	60.90	18.72	35.69	29.12	41.28	43.72
Mistral 7B-Instr (v0.3)	47.90	72.56	32.79	52.72	28.62	68.46	49.10
Mamba Codestral 7B (v0.1)	18.74	30.51	14.57	15.80	22.50	14.62	18.72
Qwen2.5-Coder 14B-Instr	51.28	80.26	51.01	57.36	30.12	62.05	41.41
Qwen3 14B	45.49	71.54	30.53	61.99	47.00	46.41	30.00
StarCoder 2 7B	36.83	68.21	21.73	43.05	15.12	53.08	48.72
Phi-4 Reasoning (14B)	21.70	26.79	35.68	15.67	45.50	18.97	8.08

Supplementary Experiments Results

Supplementary Table 5 displays the statistics of all tested models' performance over SemBench. Supplementary Table 6 displays the statistics of all tested models' performance over SemBench-Python. Given the cost limit, we only tested the preliminary Python version dataset over open-source models.

Supplementary Figure 2 shows the overall accuracy and per-category accuracy of selected models in Qwen3 and StarCoder2 families. Scaling within a family generally improves overall performance, but the trend is not strictly monotonic. For instance, dead code-statement scales smoothly within the Qwen3 family, while other categories exhibit inconsistent or non-monotone behavior. This implies that data mixture and objective weighting non-uniformly shape semantic competence, rather than a single smooth scaling law that lifts all properties equally.



Supplementary Figure 2: Qwen3 and StarCoder2 family accuracy scaled by model size

The result of this experiment, along with the nature of the semantic tasks we have constructed, proves the robustness and data-contamination-free nature of SemBench.

Correlation Computation

Supplementary Table 7: Spearman correlation between SemBench and two code–generation benchmarks. Asterisks indicate significance level: * p-value<0.05.

	HumanEval		**MBPP**		
Category	ρ	p-value	ρ	p-value	$N_{\text{HE}}/N_{\text{MBPP}}$
All Categories	0.35	0.285	0.47	0.174	11/10
DeadCode-S	0.61	0.047*	0.52	0.128	11/10
DataDep	0.33	0.326	-0.01	0.987	11/10
FuncReach	0.65	0.029*	0.73	0.016*	11/10
Dominators	0.32	0.340	0.19	0.603	11/10
DeadCode-L	0.42	0.201	0.47	0.174	11/10
Liveness	0.40	0.228	0.60	0.066	11/10

To examine whether *SemBench* performance transfers to established code-generation benchmarks, we compute rank-based correlations between each *SemBench* accuracy and the official *pass@1* reported for **HumanEval** and **MBPP**, as shown in Supplementary Table 7. Let $s_i^{(k)}$ denote the *accuracy* of model i on *SemBench* category k (including the overall SemBench), and h_i, m_i be its HumanEval and MBPP

pass@1 scores, respectively. We report

$$\rho_k^{\mathrm{HE}} = \mathrm{Spearman}\big(\{s_i^{(k)}\}, \{h_i\}\big), \qquad \rho_k^{\mathrm{MBPP}} = \mathrm{Spearman}\big(\{s_i^{(k)}\}, \{m_i\}\big),$$

together with two-sided p-values. The p-value is computed under the null hypothesis of no rank correlation ($H_0 : \rho = 0$). For a sample of size n, the Spearman correlation coefficient ρ is transformed into a t-statistic

$$t = \rho\sqrt{\frac{n-2}{1-\rho^2}},$$

which asymptotically follows a Student's t distribution with $n-2$ degrees of freedom. The reported p-value corresponds to the two-sided tail probability $\Pr(|T| \geq |t|)$.

Spearman's ρ is widely adopted for cross-benchmark studies because it is robust to non-linear scaling and outliers [12]. Models lacking an official score for either benchmark are filtered out pair-wise.

Supplementary Table 8: Correlation between SemBench and SemBench-Python across semantic categories. Asterisks indicate significance levels: * p-value<0.05, ** p-value<0.01, *** p-value<0.001.

		Spearman		**Pearson**	
Category	**N**	ρ	p-value	r	p-value
All Categories	12	0.57	0.051	0.79	0.002**
DeadCode-S	12	0.79	0.002**	0.86	<0.001***
DataDep	12	0.29	0.354	0.34	0.286
FuncReach	12	0.69	0.014*	0.65	0.022*
Dominators	12	0.38	0.226	0.57	0.053
DeadCode-L	12	0.69	0.014*	0.75	0.005**
Liveness	12	0.57	0.053	0.61	0.034*

Supplementary Table 9: Spearman correlation between SemBench-Python and MBPP. Asterisks indicate significance levels: * p-value<0.05, ** p-value<0.01.

Category	**N**	ρ	p-value
All Categories	10	0.52	0.128
DeadCode-S	10	0.77	0.009**
DataDep	10	0.26	0.467
FuncReach	10	0.82	0.004**
Dominators	10	0.27	0.446
DeadCode-L	10	0.48	0.162
Liveness	10	0.18	0.627

Supplementary Table 8 compares model performance on SemBench and SemBench-Python across the shared semantic categories. Overall, the two benchmarks show a positive cross-language relationship: the overall score has a moderate Spearman correlation ($\rho = 0.57$, $p = 0.051$) and a significant Pearson correlation ($r = 0.79$, $p = 0.002$). The strongest agreement appears in DeadCode-S, where both rank correlation and linear correlation are high and significant ($\rho = 0.79$, $p = 0.002$; $r = 0.86$, $p < 0.001$). FuncReach and DeadCode-L also show significant positive correlations, suggesting that models that perform well on these static semantic categories in C tend to perform well on their Python counterparts. DataDep and Dominators show weaker correlations, indicating that these categories may be more sensitive to language-specific program structure or to differences in the current Python construction pipeline.

Supplementary Table 9 further examines whether SemBench-Python aligns with standard code-generation performance. Since the preliminary Python evaluation covers only open-source models, we focus on MBPP, where a larger matched subset is available. The strongest correlations appear in FuncReach ($\rho = 0.82$, $p = 0.004$) and DeadCode-S ($\rho = 0.77$, $p = 0.009$), while the overall correlation is moderate but not statistically significant ($\rho = 0.52$, $p = 0.128$). This result is consistent with the SemBench finding that the relationship between static semantic understanding and code-generation ability is selective rather than uniform across semantic categories.

Because the preliminary SemBench-Python evaluation covers only open-source models, the number of models with matched official HumanEval scores is small ($N = 8$). We therefore do not report HumanEval correlations as a main result, and instead focus on MBPP, where a larger matched subset is available ($N = 10$).

Linear Regression Analysis

In addition to rank-based correlation coefficients, we visualize the relationship between *SemBench* accuracy and code-generation performance using ordinary least squares (OLS) linear regression. For each semantic category k, we consider pairs (x_i, y_i), where x_i denotes the *pass@1* score of model i on a code-generation benchmark (HumanEval or MBPP), and y_i denotes its accuracy on SemBench category k.

We fit a linear model of the form

$$y = \beta x + \alpha$$

by minimizing the squared residual error

$$\min_{\beta,\alpha} \sum_i \big(y_i - (\beta x_i + \alpha)\big)^2,$$

which yields closed-form estimates of the slope β and intercept α. The regression is computed independently for each panel in Figure 4, after filtering out models that lack an official HumanEval or MBPP score. We further report a two-sided p-value for the hypothesis test $H_0 : \beta = 0$ (no linear association), computed using the Wald t-test for the estimated slope, with $t = \hat{\beta}/\mathrm{SE}(\hat{\beta})$ and degrees of freedom $n - 2$.

The fitted line is used *only as a visual trend indicator* to summarize the direction and the approximate strength of association between benchmarks. To ensure comparability across panels, both axes are fixed to the range $[0, 100]$, and regression lines are plotted over the full horizontal domain.

Each subplot additionally reports Spearman's ρ and the corresponding two-sided p-value, annotated directly in the figure. Statistical significance is indicated using a conventional threshold ($p < 0.05$).

References

[1] Clang Static Analyzer Team. Clang Static Analyzer (2025). URL https://clang-analyzer.llvm.org/. Accessed 2025-05-14.

[2] Kremenek, T. Finding software bugs with the clang static analyzer. LLVM Developers' Meeting (Aug 2008) (2008). URL https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf. Accessed 2025-05-14.

[3] Muchnick, S. S. *Advanced Compiler Design and Implementation* (Morgan Kaufmann, 1997).

[4] Clang Team. Introduction to the clang ast (2025). URL https://clang.llvm.org/docs/IntroductionToTheClangAST.html. Accessed 2025-05-14.

[5] Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. *Compilers: Principles, Techniques, and Tools* 2nd edn (Addison-Wesley, 2006).

[6] Cooper, K. D. & Torczon, L. *Engineering a Compiler* 2nd edn (Morgan Kaufmann, 2011).

[7] LLVM Project. Llvm dominator tree pass (2025). URL https://llvm.org/docs/Passes.html#dominator-tree. Accessed 2025-05-14.

[8] LLVM Project. scalar-evolution: Scalar evolution analysis (2025). URL https://llvm.org/docs/Passes.html#scalar-evolution-analysis. LLVM Documentation (*LLVM's Analysis and Transform Passes*); Accessed 2025-05-14.

[9] LLVM Project. LLVM Project – Version 17.0.6 (2024). https://llvm.org/ (Accessed 2025-05-14).

[10] Clang Team. Clang 17.0.6: A C/C++/Objective-C Front End for LLVM (2024). https://clang.llvm.org/ (Accessed 2025-05-14).

[11] Xie, D. *et al.* *CoRe: Benchmarking LLMs' code reasoning capabilities through static analysis tasks* (2025). URL https://openreview.net/forum?id=WJIDorHiuZ. NeurIPS 2025 Datasets and Benchmarks Track spotlight.

[12] Liang, P. *et al.* Holistic evaluation of language models. *Transactions on Machine Learning Research* (2023). URL https://openreview.net/forum?id=iO4LZibEqW. Also available as arXiv:2211.09110.