Distributional Program Analysis: Treating Large Language Models as Program Samplers

Phat T. Tran-Truong
Ho Chi Minh City University of Technology (HCMUT),
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
phatttt@hcmut.edu.vn

Xuan-Bach Le✉
Ho Chi Minh City University of Technology (HCMUT),
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
lexuanbach@hcmut.edu.vn

Abstract

A specification under-determines a program: prompted with one, a large language model (LLM) induces a distribution over candidate programs that respond to it in different ways. Pipelines that generate one program and analyze it discard the key artifact of that distribution: *variation across samples*. We propose **distributional program analysis** (DPA): analyze many LLM samples for the same task and aggregate analyzer outputs as cross-sample evidence about candidate program properties. The central statistic is an invariant's *survival rate*, the fraction of samples in which the analyzer infers it. Under stated sampling and analyzer assumptions, high-survival invariants mark properties on which independently sampled programs agree; fragile ones may indicate decoding noise, underspecification, or analyzer blind spots. Agreement is not correctness, and we are explicit about when the two come apart. We define survival, release a prototype, and report a HumanEval+ pilot in which high-survival invariants match reference-solution invariants in roughly four of five cases; filtering by survival also improves pass@1 at the same 20-sample budget. Variation is an analysis signal, not noise to hide.

CCS Concepts

• **Software and its engineering** → **Automated static analysis**; *Software verification and validation*; • **Computing methodologies** → *Natural language generation.*

Keywords

program analysis, large language models, likely invariants, invariant inference, uncertainty estimation, code generation

ACM Reference Format:
Phat T. Tran-Truong and Xuan-Bach Le. 2026. Distributional Program Analysis: Treating Large Language Models as Program Samplers. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834555

1 Introduction

A specification rarely identifies a single program: two correct implementations of `is_prime` differ in branching, representation, and boundary behavior. Fed such a specification, a large language model (LLM) samples from a distribution of plausible programs—some implementing the intended behavior, others near misses that reveal what the prompt left unsaid. Most pipelines pairing LLM generation with static analysis collapse that distribution at once: they pick one sample, analyze it, and accept or reject it. That discards the strongest signal available. An analyzer run on one generated program describes that artifact; the same analyzer across many samples characterizes the distribution the specification induces, including where it is stable and where it is ambiguous.

We make one conceptual move: treat the LLM as a *program sampler* and the analyzer as a property extractor. Given N samples for a specification ϕ, the analyzer yields candidate invariants—ranges, nullability facts, length and monotonicity relations—and for each we compute its *survival rate*: the fraction of samples in which it is inferred. Under explicit assumptions about sample diversity and analyzer expressiveness, an invariant surviving in 95% of samples is stronger evidence about what ϕ pins down than one surviving in 30%.

We call this framing **distributional program analysis** (DPA). The paper defines invariant survival, implements it in a released prototype, and evaluates the idea in a HumanEval+ pilot [14]. In the pilot, survival correlates with whether an invariant appears in the reference solution ($r \approx 0.78$), and filtering samples by agreement with high-survival invariants improves pass@1 at a matched 20-sample budget relative to majority voting.

Contributions. (i) A framing in which an LLM is a program sampler, an analyzer extracts candidate semantic properties, and *invariant survival* summarizes cross-sample evidence about what a specification determines, under stated assumptions (Section 3). (ii) A formal definition of survival-based property weighting and a released single-task prototype whose analyzer, vocabulary, and canonical form are stated in full (Section 3). (iii) A HumanEval+ pilot (Section 4) in which survival aligns with reference-solution invariants and improves pass@1 at a matched sample budget, and an evaluation roadmap (Section 5) naming the controls that would let survival carry a correctness claim.

2 A Motivating Example

Consider the prompt: *"Implement* `rolling_max(xs, k)` *that returns the maximum value in each length-k window of* `xs`*."* We work throughout with the fixed 20-sample suite released with our artifact, spanning correct implementations and near misses; 17 of the 20 satisfy the reference oracle on all seven probe inputs. Listing 1 contrasts two, and others use monotonic-deque or explicit-loop formulations behind the same interface.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834555

```python
# Sample A -- right-aligned windows
return [max(xs[i:i+k]) for i in range(len(xs)-k+1)]
# Sample C -- left-aligned prefix windows
return [max(xs[max(0,i-k+1):i+1]) for i in range(len(xs))]
```

Listing 1: Return expressions of two released samples, both guarded by `if k <= 0: return []`. A passes the oracle, C does not.

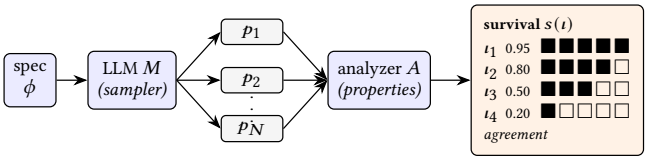

Figure 1: The DPA framing. The LLM is a *sampler* over programs satisfying ϕ; the analyzer yields a candidate-invariant set per sample; survival rates summarize cross-sample agreement. Survival values shown are illustrative; Table 1 reports measured ones.

The analyzer of Section 3.5 infers eight distinct facts across the suite (Table 1). Five survive in every sample, including `len(result) <= len(xs)`. Two sharper facts do not: `len(result) == max(0, len(xs) - k + 1)` survives 18/20, and the full postcondition `result[i] == max(xs[i:i+k])` survives 17/20. Sample C violates both, because its left-aligned window returns one entry per input element.

The disagreement is the signal. Here it marks an error rather than an ambiguity—the prompt asks for length-k windows, so Sample C's prefix windows are a near miss, not a competing convention—but survival exposes the divergence either way, and the same profile would surface a boundary the prompt genuinely left open. Where a single-sample pipeline validates whichever variant it drew, a distributional one flags the minority for filtering and the divergence for clarification.

3 Distributional Program Analysis

3.1 Setup

Figure 1 summarizes the framing. Let ϕ be a specification: a natural-language prompt, a partial test suite, or both. Let M be an LLM, and write $\mathcal{D}_\phi = M(\cdot \mid \phi)$ for the induced distribution over generated programs, following the view of code as a structured statistical object [15, 23]. Where ϕ is realized by several paraphrases, ϕ denotes the paraphrase policy itself—$q \sim Q(\cdot \mid \phi)$, then $p \sim M(\cdot \mid q)$—and $\mathcal{D}_\phi$ is the induced mixture. Sampling N times yields $p_1, \ldots, p_N$. Let A be a deterministic property extractor, including its canonicalizer, mapping p to a set $\mathcal{I}(p)$ of *candidate invariants*: boolean predicates over input/output pairs or over internal state at named program points.

3.2 Survival as Cross-Sample Evidence

Let $\mathcal{I}^* = \bigcup_{i=1}^{N} \mathcal{I}(p_i)$ be the union of candidate invariants observed across all samples. For each $\iota \in \mathcal{I}^*$, define its *survival rate* as

$$s_N(\iota) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[\, \iota \in \mathcal{I}(p_i) \,].$$

We write $s(\iota)$ when N is fixed by context. Its population counterpart is $\pi_\iota = \mathbb{P}_{p \sim \mathcal{D}_\phi}[\iota \in \mathcal{I}(p)]$, the probability that A reports ι on a fresh sample from $\mathcal{D}_\phi$. So $s_N(\iota)$ estimates property recurrence in the generated-program distribution, not correctness of one program—the stance of statistical model checking, which draws evidence about a stochastic system from sampled executions [25]. Were samples independent, Hoeffding-style concentration would make large gaps between $s_N(\iota)$ values evidence about gaps between π_ι; for LLM samples that is a reference point rather than a guarantee, and the full study measures effective sample size under correlated decoding.

DPA relies on five operating conditions. (A1) Sampling is diverse: $\mathcal{D}_\phi$ has not collapsed to near-duplicates. (A2) Model errors are not perfectly correlated across samples. (A3) The analyzer and canonicalizer are deterministic for the same program. (A4) The invariant language can express task-relevant properties. (A5) Equivalent facts are canonicalized before counting survival. Under these conditions, high survival is evidence of recurring structure; violations produce false consensus, analyzer blind spots, or canonicalization artifacts.

We therefore call an invariant with $s(\iota) \geq \tau$ a *consensus invariant*, and use that term rather than "intended" throughout. Consensus is a property of the sampled distribution, not of the developer's intent: if a misconception is shared across samples—(A2) fails—a wrong invariant can reach $s(\iota) = 1$ and survival reports the shared error with full confidence. Survival is evidence about what ϕ pins down under M and A, evidence about correctness only insofar as (A1)–(A5) hold, and never a substitute for verification.

3.3 Decisions DPA Enables

The survival profile $s(\cdot)$ supports three decisions. *Sample filtering* rejects samples violating any invariant with $s(\iota) \geq \tau$ ($\tau = 0.85$ in the pilot): they disagree with the dominant property profile. If no sample satisfies every such invariant the filter abstains, returning the unfiltered set. *Specification refinement* surfaces the highest-survival invariants as *discovered* pre/post-conditions, candidates for explicit specification, echoing partial specifications in specification mining and oracle-guided synthesis [1, 8]. *Disagreement triage* treats mid-range survival (heuristically $0.4 < s(\iota) < 0.7$) as a prompt for inspection: the fact may mark an under-specified design choice, but equally an outright error or an analyzer artifact. Surfacing it turns one-shot generation into a clarification dialogue.

3.4 Pipeline

Our Python prototype performs five steps. (1) Sample $N = 20$ programs from the LLM at temperature 0.8 with varied prompt paraphrases. (2) Run the property extractor of Section 3.5 on each sample. (3) Canonicalize the resulting facts. (4) Aggregate survival counts. (5) Apply the three DPA decisions above on top of a downstream pipeline such as test selection, voting, or repair. Only step

(2) is analyzer-specific: A is a framework parameter, and any deterministic extractor with a canonical output form can be substituted.

3.5 Invariant Language and Canonicalization

Survival can only see what A can express, so we state the released extractor precisely. It is a *deterministic checker over a closed predicate vocabulary*, in the likely-invariant tradition of Daikon [4], not a static abstract interpreter. A candidate invariant is a pair $\langle kind, body \rangle$ over the formal parameters and returned value. There are five kinds: `type`, `boundary`, `elementwise`, `postcondition`, and `length_relation`. The *body* may carry a guard, written ι `when` g. A fact enters $\mathcal{I}(p)$ only if it holds on *every* probe without raising, so one exception suppresses it; Table 1 lists the eight facts of the running example.

Two consequences cut against overclaiming. A closed vocabulary makes canonicalization exact—facts compare by their $\langle kind, body \rangle$ pair, so (A3) and (A5) hold by construction—but bounds expressiveness to what the vocabulary contains, the binding limit on DPA today. And facts established by probing rather than proof are *likely* invariants: survival compounds cross-sample agreement with probe coverage, so a property no probe distinguishes stays invisible.

3.6 Applicability and Cost

Cost and scale. The relevant comparison is not one analysis but self-consistency-style aggregation [27], which already spends N samples. The marginal cost over that baseline is one analyzer run per sample, under 5% of inference wall-clock in the pilot. Two structural facts keep it there: the analyses are independent, so they parallelize without coordination—a property of the problem, not of our serial runner—and each covers one generated function rather than a whole program, so cost grows linearly in N. DPA does not make an expensive whole-program analysis cheap; where one run is already the bottleneck, N runs are unaffordable, and scaling to repositories is a separate problem attacked by compressing the representation and validating assumptions explicitly [11].

Analyzer availability. DPA needs an analyzer producing canonicalizable facts in the language of Section 3.5. Where none exists, DPA does not apply, bounding the approach more tightly than the sampling assumptions do. Where a weak one exists, DPA degrades rather than fails: samples yielding no usable facts stay in the denominator of $s_N(\iota)$, so limited coverage appears as uniformly depressed survival, never as silently discarded samples. Our vocabulary sits near that floor—a bet the full study tests by ablating further.

3.7 Artifact Evidence

`make smoketest` reproduces this subsection deterministically: it runs the extractor over the released suite, writes per-sample invariants and survival rates to `runs/`, and applies DPA-filter at $\tau = 0.85$. Table 1 is its complete output—eight facts, 7.7 per sample. Filtering keeps 17 of 20 samples, removing exactly the three that fail the oracle, so survivor purity is 100% against a raw rate of 85%.

Two ablations run on the same suite. Raising the threshold degrades the filter monotonically: purity is 100% for every $\tau \leq 0.85$, then 94.4%, 89.5%, and 85.0% at $\tau = 0.90$, 0.95, and 1.00, where only the five unanimous facts stay required and filtering becomes vacuous. The operating point is flat below 0.85 and decays above it, not a knife edge. Dropping three samples at random instead yields a fully correct pool in 10 of 10,000 draws (0.1%) against 100% for DPA-filter, so the gain comes from the invariant signal, not from shrinking the pool. Analysis costs 0.18 ms per sample.

Table 1: Every fact inferred on the 20-sample `rolling_max` suite. A guard restricts a fact to inputs meeting it.

Invariant	Guard	$s(\iota)$
`result is list`		20/20
`len(result) <= len(xs)`		20/20
`all(result[i] in xs)`		20/20
`all(result[i] >= min(xs))`	`xs`	20/20
`result == []`	`k <= 0`	20/20
`result == []`	`len(xs)<k`	19/20
`len(result) == max(0,len(xs)-k+1)`	`k > 0`	18/20
`result[i] == max(xs[i:i+k])`	`k > 0`	17/20

One task with a fixed suite shows the mechanism works and is reproducible, not that it generalizes; and the suite holds 17 distinct sources among 20 samples, a mild instance of the collapse (A1) excludes and Q3 measures.

4 Pilot Study

Setup. We sampled $N = 20$ programs per task from a frontier LLM on the 164 tasks of HumanEval+ [14], using temperature 0.8 and three prompt paraphrases. For each task we ran the analyzer, computed survival rates, and compared them to a hand-extracted set of reference-solution invariants: pre/post-conditions and loop-relevant facts from the canonical solution, canonicalized by the same rules as analyzer outputs. It is an operational proxy for intent, and we treat it as one below.

Result 1: survival aligns with reference-implementation invariants. Pearson correlation between $s(\iota)$ and the indicator "ι appears in the reference solution" was $r = 0.78$ ($p<0.001$). Among invariants with $s(\iota) \geq 0.85$, 82% are present in the reference solution; among those with $s(\iota) \leq 0.30$, only 11% are. The signal is informative at both ends of the range.

We are explicit about what this does not measure. The canonical solution is one correct implementation, not the intended specification: a correct alternative may satisfy different valid invariants, so part of $r = 0.78$ is agreement with that implementation's *style*, not with intent. The 18% of high-survival invariants absent from the reference are the interesting cases, and we cannot yet separate the wrong ones from those valid under an alternative; Q3 names the control that does. Until it runs, Result 1 supports only the weaker claim that survival tracks the invariants of *a* correct implementation.

Result 2: filtering improves pass@1 at fixed compute. Each aggregator selects one program per task from the same 20 samples; pass@1 is the fraction of tasks whose selection passes the full suite, i.e. selection accuracy, not the unbiased pass@k estimator. Table 2 compares pick-first (a random sample), self-consistency (majority

Table 2: HumanEval+ pilot. All conditions use the same 20 LLM samples per task. Pass@1 rates over 164 tasks.

Aggregator	pass@1
Pick-first	64.6%
Self-consistency [27]	71.3%
DPA-filter (ours)	**75.0%**

vote on test outputs), and DPA-filter (drop samples violating any $s(\iota) \geq 0.85$ invariant, then majority vote).

The 3.7-point absolute lift over self-consistency at a *matched sample budget* is the central preliminary finding. It concentrates on tasks with under-specified boundary behavior, where symptom-level voting fails: wrong samples may agree on a corner-case output while analyzer-derived invariants disagree on the function's structure and break the tie. On the 18 such tasks DPA solved and self-consistency missed, the surviving invariants captured exactly the ambiguous convention—off-by-one indexing, empty-list return, exception versus silent default.

5 Future Plans

Q1: Does the lift transfer, and against which baselines? We will replicate on MBPP+ [14], LiveCodeBench [7], and a held-out corpus, to test whether DPA exploits a general property of LLM code distributions or a HumanEval-family artifact. Section 3.7 separates DPA-filter from random filtering on one task; at scale we add CodeT-style test filtering [2], LEVER-style execution reranking [19], behavior-cluster voting, and an analyzer score over the first sample.

Q2: Is DPA a substitute or a complement to capability? We will sweep a frontier closed model and two open-weight models, expecting the largest lift on the smallest, which leaves the most decoding noise to filter. The deployment we envisage is an IDE-time check showing a function's consensus invariants beside its docstring.

Q3: When does consensus stop being evidence? Three controls target the failure modes above. Against a weak intent proxy, we re-extract reference invariants from a *second* correct implementation per task and report how many high-survival invariants missing from the canonical solution hold under it. Against collapse, we measure structural diversity and report DPA performance against it. Against false consensus, we compare within-family with cross-family survival. We also report threshold sensitivity, confidence intervals, and effective-sample-size diagnostics.

6 Threats to Validity

Internal validity. Survival estimates analyzer-visible property recurrence; at $N = 20$ sampling variance and correlation matter, so paired confidence intervals, effective sample size, and τ/N sensitivity will calibrate the full evaluation.
Construct validity. The dominant threat is *false consensus*: when (A2) fails and samples share a misconception, the wrong invariant survives everywhere and is reported with maximal confidence, and nothing within one model family separates that from genuine agreement—so the cross-family control of Section 5 precedes any correctness claim. Reference-solution invariants only proxy for intent, and DPA sees only what Section 3.5 expresses.
External validity. The pilot uses one Python analyzer on HumanEval-family tasks and Section 3.7 is a single task; we will ablate the analyzer and replicate on LiveCodeBench and MBPP+.
Conclusion validity. Contamination affects the lift; threshold choice does not, on the task we can sweep. The full study adds contamination controls, paired uncertainty, and threshold sweeps.

7 Related Work

LLM aggregation. Self-consistency [27] and majority voting aggregate outputs; DPA aggregates inferred properties across implementations, separating programs that agree on visible tests but differ in loop bounds, boundary conventions, or error handling. AlphaCode filtering [13], CodeT [2], and LEVER [19] also select among samples, but through test agreement or execution rather than inferred properties.

Invariant inference and sampling-based evidence. Abstract interpretation [3] over-approximates program behavior; Daikon infers *likely* invariants from executions [4, 5], and specification mining extracts candidates from traces [1]. DPA inherits that caution—mined invariants are evidence, not proof—and changes the axis, mining across many implementations of one specification. Statistical model checking [25] is the closest sampling analogue, though survival carries no hypothesis test; probabilistic programming [26] and probabilistic abstract interpretation [16] instead give semantics to probabilistic computation. By analogy, disagreement among independently trained models [10] motivates the cross-family control of Section 5. Statistical code models [15], analysis-aware completion [23], and generation modulo static analysis [17] couple code distributions to analyzers *before* generation; DPA aggregates after it.

Analysis of neural code, and incomplete specifications. Studies of LLM-generated code evaluate many generated programs, but judge each one individually [20, 24]; DPA makes the variation between them the object; surveys chart how LLM agents are entering static analysis [6] and how explanations are surfaced across the software lifecycle [12]. SPoC [9] searches candidate translations under test feedback and Synchromesh [21] constrains decoding itself; semantic repair [18] is a future baseline. DPA instead filters through inferred properties. Where oracle-guided [8] and refinement-type synthesis [22] consume specifications, DPA mines candidates that may become them.

8 Conclusion

DPA treats the distribution an LLM induces as evidence: survival is a compact signal of agreement among sampled programs under stated assumptions—not of correctness, and not of intent. The pilot shows it aligns with reference-solution invariants and improves filtering at a matched budget. The next step is to find where survival keeps that role and where consensus turns false.

Acknowledgments

We acknowledge Ho Chi Minh City University of Technology (HCMUT), VNUHCM for supporting this study.

Data Availability Statement

The prototype, the fixed 20-sample suite, and the recorded runs behind Section 3.7 are archived at https://doi.org/10.5281/zenodo.21863484, where `make` regenerates Table 1 and both ablations. HumanEval+ pilot data will follow with the full study.

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 4–16. doi:10.1145/503272.503275

[2] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *International Conference on Learning Representations (ICLR)*.

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*. 238–252. doi:10.1145/512950.512973

[4] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. doi:10.1109/32.908957

[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45. doi:10.1016/j.scico.2007.01.015

[6] Hoang-Quoc-Bao Hua and Xuan-Bach Le. 2026. Emerging Trends and Challenges Toward LLM Agents in Static Analysis. In *Proceedings of the 39th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*. doi:10.1007/978-981-92-2885-0_9

[7] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *International Conference on Learning Representations (ICLR)*.

[8] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 215–224. doi:10.1145/1806799.1806833

[9] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-Based Pseudocode to Code. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[10] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[11] Thi-Hong-Cuc Le, Hoang-Quoc-Bao Hua, and Xuan-Bach Le. 2026. GraphLedger: Repository-Level Vulnerability Detection via Graph-Based Compression and Assumption Validation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1145/3832783.3834420

[12] Thi-Hong-Cuc Le and Xuan-Bach Le. 2026. From Black-Box to Glass-Box: Explainable AI for Applied Intelligent Software Systems Across the Software Development Life Cycle. In *Proceedings of the 39th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*. doi:10.1007/978-981-92-2891-1_8

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158

[14] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.52202/075280-0943

[15] Chris J. Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*. 649–657.

[16] David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *Proceedings of the 7th International Symposium on Static Analysis (SAS)*. 322–339. doi:10.1007/978-3-540-45099-3_17

[17] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. 2021. Neural Program Generation Modulo Static Analysis. In *Advances in Neural Information Processing Systems (NeurIPS)*. 18984–18996.

[18] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 772–781. doi:10.1109/ICSE.2013.6606623

[19] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML) (PMLR, Vol. 202)*. 26106–26128.

[20] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. doi:10.1109/SP46214.2022.9833571

[21] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-Trained Language Models. In *International Conference on Learning Representations (ICLR)*.

[22] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 522–538. doi:10.1145/2908080.2908093

[23] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 419–428. doi:10.1145/2594291.2594321

[24] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *Proceedings of the USENIX Security Symposium*.

[25] Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2004. Statistical Model Checking of Black-Box Probabilistic Systems. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*. 202–215. doi:10.1007/978-3-540-27813-9_16

[26] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *arXiv preprint arXiv:1809.10756* (2018). doi:10.48550/arXiv.1809.10756

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *International Conference on Learning Representations (ICLR)*.

Received 2026-05-13; accepted 2026-07-02