Probabilistic Programs of Thought

Poorva Garg
University of California Los Angeles
poorvagarg@cs.ucla.edu

Renato Lui Geh
University of California Los Angeles
renatolg@cs.ucla.edu

Daniel Israel
University of California Los Angeles
daniel.m.israel@cs.ucla.edu

Todd Millstein
University of California Los Angeles
todd@cs.ucla.edu

Kyle Richardson
Allen Institute for AI
kyler@allenai.org

Guy Van den Broeck
University of California Los Angeles
guyvdb@cs.ucla.edu

Abstract

LLMs are widely used for code generation and mathematical reasoning tasks where they are required to generate structured output. They either need to reason about code, generate code for a given specification, or reason using programs of thought. The typical approach to code generation is to prompt the model and generate samples until an appropriate program is obtained. Within this process, sampling n programs from the language model requires n GPU compute-intensive generations which becomes prohibitively expensive for larger values of n. In this work, we address this limitation by exposing the LLM's distribution within the generated programs themselves. We propose a novel test-time framework we dub *probabilistic programs of thought* to obtain more samples from the model with fewer LLM generations. Given a program generated by a model and the associated next-token probabilities, we build a probabilistic program that compactly represents exponentially many deterministic programs. Since performing probabilistic reasoning in this probabilistic program is much cheaper, our approach allows sampling new programs without any additional GPU compute and little CPU overhead. We instantiate our approach on benchmarks for code generation, code understanding and mathematical reasoning and report improvements in performance with fewer generations from the LLM.

1 Introduction

Large language models (LLMs) are increasingly being used for coding tasks requiring them to understand, edit, and generate code according to a given specification (Jiang et al., 2026; Gu et al., 2024a). Additionally, generating code has proven to be a useful means for language models to reason through complex mathematical problems under the paradigm of *programs of thought* (Chen et al., 2023).

The typical workflow for code generation involves prompting the language model with the task, getting back a code snippet, and executing it to verify. If the program is correct, it is incorporated, otherwise the language model is prompted again. This process is repeated until a (sufficiently) correct program is found and is often equipped with more elaborate test-time decoding strategies such as Best-of-n (Cobbe et al., 2021a) and beam search (Snell et al., 2024). However, this sample-execute-verify loop requires multiple programs and as such multiple generations from the LLM, making the process prohibitively GPU compute-intensive in larger models for even a small number of samples.

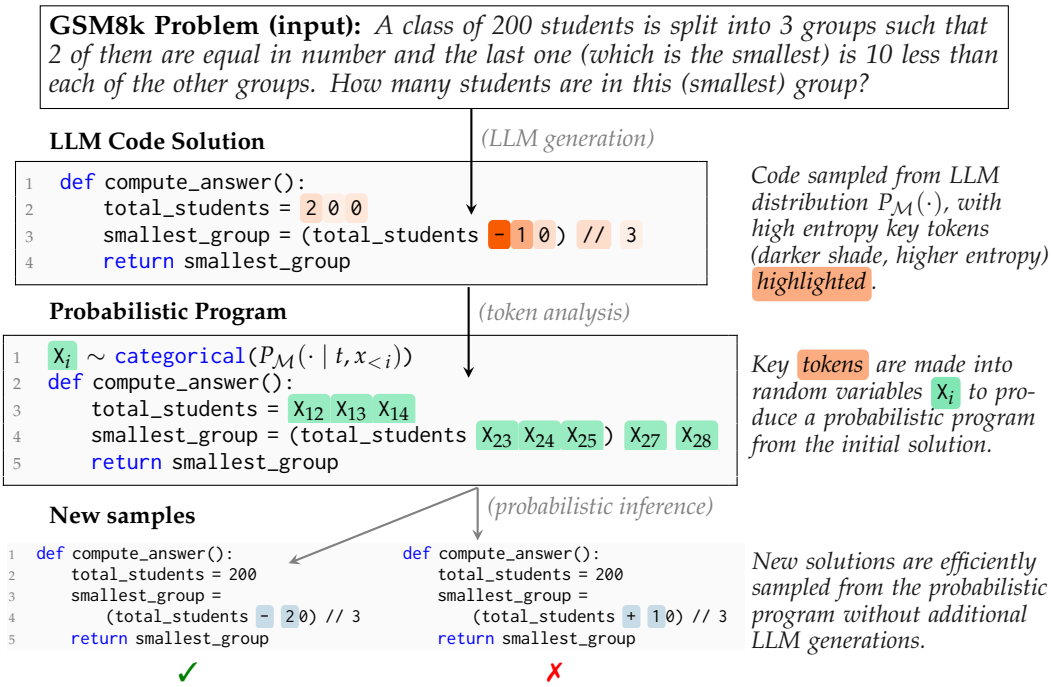


Figure 1: **A probabilistic program of thought can sample a correct program from an incorrect program of thought.** An LLM generated program is equipped with probability distributions parameterized from the logits of the LLM; select tokens are treated as random variables X_i, turning the code into a probabilistic program.

In this paper, we propose addressing these limitations by exposing the LLM's underlying probability distribution within each generated program. We present *probabilistic programs of thought* (PPoT): a novel test-time decoding technique for code generation that uses the perspective of probabilistic programming—a paradigm where programming constructs are coupled with probability distributions in order to build statistical models—to perform probabilistic reasoning. This is arguably a very natural perspective to LLM code generation: an LLM already inherently models a distribution over a program's components (i.e. constants, operators, functions, etc.) albeit at a lower (token) level. Since the resulting probabilistic program can be sampled inexpensively, it enables generating multiple samples using only a single LLM call.

Figure 1 illustrates our approach. Given a mathematical reasoning problem, the language model generates a deterministic program whose tokens can then be converted into random variables, turning the deterministic program into a probabilistic program[1]. We interpret tokens corresponding to digits and arithmetic operators as random variables whose distributions are parameterized by the language model's next-token logits. This results in a probabilistic program that incorporates the uncertainty of the LLM and can sample new correct programs without any further GPU compute.

Overall, our contributions are as follows:

1. **We propose *probabilistic programs of thought* (PPoT)**, a novel decoding technique for structured output generation that allows for more samples to be efficiently generated.
2. **We show our approach to be distributionally sound** with respect to the language model distribution under certain independence assumptions (Section 3.2).
3. **We empirically verify that PPoT achieves consistent improvement** in pass@k performance, with an up to 7% increase in accuracy on Qwen2.5 (Hui et al., 2024; Bai et al.,

[1]In this paper, we write probabilistic programs in an unusual syntax, namely as probabilistic strings of program text. It is straightforward to translate this syntax into a more classic probabilistic program syntax using coin flips, if-then-else with random variable guards, etc.

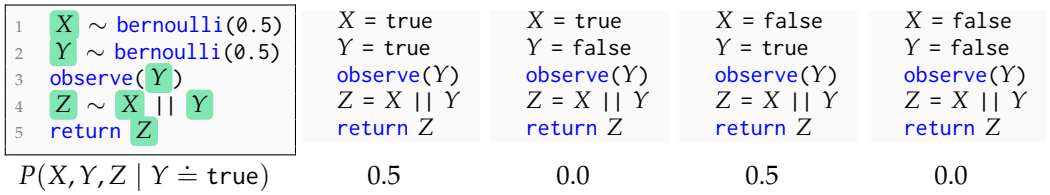


Figure 2: **A probabilistic program (left) induces a distribution over exponentially many deterministic programs (right).** This probabilistic program defines random variables X, Y and Z, conditions on $Y \doteq \texttt{true}$ and returns Z as the logical or of X and Y; each deterministic program (and their execution) thus has a probability according to this distribution.

2025) across three different benchmarks—GSM8k (Cobbe et al., 2021b), Plot2Code (Wu et al., 2024), and CRUXEval (Gu et al., 2024b)—without any additional generations from the LLM (Section 4.1).

4. **We provide quantitative evidence that PPoT exhibits significant computational savings** over directly sampling from the LLM (Section 4.2).

2 Code Generation as a Probabilistic Program

We start this section by providing some necessary background on probabilistic programs and describe how they capture distributions over deterministic programs[2]. We then observe how code generation with a language model can in fact be naturally viewed as a probabilistic program, a view which we then exploit to develop our novel decoding technique.

2.1 What are Probabilistic Programs?

Probabilistic programming languages (PPLs) (Gordon et al., 2014; Katoen et al., 2015; Pfeffer, 2016) extend ordinary programs with two constructs: (1) the ability to define random variables drawn from probability distributions (e.g., via `bernoulli(.)` and `categorical(.)`) and (2) the ability to condition on variable values through observations (via `observe(.)`). These two additions suffice to declaratively specify complex statistical models while separating modeling from inference.

We shall denote random variables (RVs) by upper case letters or capitalized words (e.g. X, Y, Z, Exc, Prog), values by lower case (e.g. x, y, z, exc, prog), and sets and sequences with bold font (e.g. $\boldsymbol{X}$ for RVs, $\boldsymbol{x}$ for values). RVs will be highlighted in green when denoting them in a probabilistic program (e.g. X, Y, Z). The notation $\boldsymbol{X}_{<n}$ will be used to mean the subsequence $(X_i)_{i=1}^{n-1}$. Figure 2 illustrates a simple probabilistic program that defines two independent Bernoulli random variables X and Y, conditions on $Y \doteq \texttt{true}$ (via `observe(.)`), and returns Z defined as $X||Y$. Under standard semantics (Dahlqvist et al., 2020; Holtzen et al., 2020), this program captures the following distribution.

$$P(X, Y, Z \mid Y \doteq \texttt{true}) = \frac{P(X) \cdot P(Y) \cdot [\![Z \doteq X||Y]\!] \cdot [\![Y \doteq \texttt{true}]\!]}{P(Y \doteq \texttt{true})} \tag{1}$$

where $[\![\cdot]\!]$ is the Iverson bracket and $\doteq$ denotes equality testing. Crucially, this single probabilistic program compactly represents a distribution over deterministic programs exponential in the number of random variables (e.g. Figure 2 left encodes 2^2 deterministic programs as shown on the right). This demonstrates how expressive probabilistic programs can be which motives our use of probabilistic programming for code generation in the following sections.

[2]Typically, probabilistic programs are described as capturing distributions over program executions. In the context of this work, we consider different executions as different deterministic programs.

2.2 LLMs as Probabilistic Programs

Code generation from an autoregressive LLM proceeds by sampling a token sequence from an autoregressive distribution, parsing it as a program, and executing it. Repeated sampling yields samples from the distribution that that LLM induces over programs (denoted by Prog) and their execution results (resp. Exc). We make this precise by casting this process as a declarative probabilistic program.

Definition 1 (LLM Code-Generating Probabilistic Program). *Given a prompt $\boldsymbol{t}$, the distribution $P_{\mathcal{M}}$ of a language model $\mathcal{M}$ induces a distribution P_{code} over the space of deterministic programs and execution results. P_{code} can be defined as the distribution of the following probabilistic program where $P_{\mathcal{M}}(\cdot \mid \boldsymbol{X}_{<i}, \boldsymbol{t})$ refers to the next-token probabilities*

```
X_1 ~ categorical(P_M(·|t))
X_2 ~ categorical(P_M(·|X_1, t))
X_3 ~ categorical(P_M(·|X_1, X_2, t))
...
X_n ~ categorical(P_M(·|X_1, X_2, ..., X_{n-1}, t))
Prog ~ parse(X_1, X_2, X_3, ..., X_n)
Exc ~ execute(Prog)
return Prog, Exc
```

Each token X_i in the program above is a categorical random variable parameterized by the next-token distributions $P_{\mathcal{M}}(\cdot \mid \boldsymbol{X}_{<i}, \boldsymbol{t})$. The joint distribution induced by this program is:

$$P_{code}(X_1, X_2, \ldots X_n, \mathsf{Prog}, \mathsf{Exc} \mid \boldsymbol{t}) = P_{\mathcal{M}}(X_1, X_2, \ldots X_n \mid \boldsymbol{t}) \cdot [\![\mathsf{Prog} \doteq \texttt{parse}(X_1, \ldots X_n)]\!] \cdot [\![\mathsf{Exc} \doteq \texttt{execute}(\mathsf{Prog})]\!]. \quad (2)$$

In the above equation, the indicator function for the equality constraint program $\mathsf{Prog} \doteq \texttt{parse}(X_1, X_2, \ldots, X_n)$ ensures that the distribution is non-zero only over valid programs that can be obtained from the token sequence. Same is the case for the other equality constraint $\mathsf{Exc} \doteq \texttt{execute}(\mathsf{Prog})$. A key benefit of this formulation is that it naturally supports describing further computation over P_{code} in a declarative fashion without needing to specify details of inference. For instance, conditioning on a target output exc is expressed via $\texttt{observe}(\mathsf{Exc} \doteq \mathsf{exc})$. Lew et al. (2023) takes a similar approach to constrain natural-language output from language models.

3 Probabilistic Programs of Thought

We introduced the code-generating probabilistic program in the previous section that captures the distribution that a language model induces on the generated programs. This section describes how probabilistic inference can be performed for these probabilistic programs. We first describe the typical code generation process as an inference algorithm, and then we describe our approach: *probabilistic programs of thought* (**PPoT**), an alternate compute-efficient inference algorithm.

3.1 Typical Code Generation

A common inference strategy for sampling from distribution P_{code} is to autoregressively sample *concrete* sequences from the model, parse them as programs and execute them to get the results. This corresponds to the programs-of-thought approach of Chen et al. (2023). Under this approach, one only ever parses and executes concrete deterministic programs. Algorithm 1 provides a procedural description for how to obtain k samples from P_{code}. We use the `sample` function to denote sampling a concrete value from the distribution of a random variable. Note that this typical programs-of-thought approach requires k requests to the language model to obtain k independent samples from P_{code}.

Algorithm 1 Naïve Inference Strategy: Programs-of-Thought

Input Prompt $\boldsymbol{t}$, sequence length n, number of samples k

```
samples ← []
for j ∈ {1, 2, ..., k}
    for i ∈ {1, 2, ..., n}
        x_i ← sample(P_M(. | x_<i, t))
    prog ← parse(x_1, x_2, ... x_n)
    exc ← execute(prog)
    samples ← samples + [(prog, exc)]
return samples
```

3.2 Probabilistic Programs of Thought

Algorithm 1 requires repeated calls to the LLM to generate k samples because it throws away the next-token probabilities once sampling is finished. Our approach, probabilistic programs of thought (PPoT) is based on the insight that we can reuse these next-token probabilities to generate more samples without querying the language model again. In particular, we propose to push forward (Holtzen et al., 2018; Bogachev, 2007, Section 3.6) the language model distribution through the `parse` function and obtain a probabilistic program instead of a deterministic program. The intuition is that these probabilistic programs act as rough approximations of the probabilistic program encoding the LLM distribution. PPoT can in fact be seen as a tractable representation of (part of) this distribution. We explain this through the following example:

Prompt: Write a function to add the smallest positive even number and the smallest prime number.
LLM output: "`def solution(): return 0 + 2`"
Parsed program (prog): `def solution(): return 0 + 2`
Execution (exc): 2

Here, the language model incorrectly outputs `0` as the smallest positive even number yielding incorrect result. This requires generation of a new sample and we propose to do so without sampling again from the language model. Note that in the language model output, the token corresponding to `0` does not need to be sampled concretely before parsing. Instead it can simply be treated as a categorical distribution parameterized by the next token probabilities from the language model. This results in the parsed Prog becoming a probabilistic program that looks as follows:

```
X ~ categorical(P_M(. | "def solution(): return ", t))
def solution(): return X + 2
```

Note that this probabilistic program Prog captures part of the LLM distribution around the LLM output "`def solution(): return 0 + 2`" and is much easier to reason about than the language model itself. We use Prog to obtain more samples without querying the language model repeatedly. To provide further intuition, we provide the adapted version of Equation (2) below. If $\boldsymbol{L}$ is the set of indices that we turn into random variables (e.g. the set with only the index of token `0` in the above example), then

$$P_{code}(X_1, X_2, \ldots X_n, \mathsf{Prog}, \mathsf{Exc} \mid \boldsymbol{t}) = \underbrace{\prod_{i \notin \boldsymbol{L}} P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i}, \boldsymbol{t})}_{\text{Sampled from LM}} \underbrace{\prod_{j \in \boldsymbol{L}} P_{\mathcal{M}}(X_j \mid \boldsymbol{X}_{<j}, \boldsymbol{t}) \cdot [\![\mathsf{Prog} \doteq \texttt{parse}(X_1, \ldots, X_n)]\!] \cdot [\![\mathsf{Exc} \doteq \texttt{execute}(\mathsf{Prog})]\!]}_{\text{Probabilistic program—cheaper to sample from repeatedly}} \quad (3)$$

In accordance to the equation above, we first sample a program from the language model. Based on this sample we identify a set of token indices $\boldsymbol{L}$ that we turn into random variables (more detail in Section 3.3). We keep tokens whose indices do not belong in $\boldsymbol{L}$. This gives us a sample from the distribution $\prod_{i \notin \boldsymbol{L}} P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i}, \boldsymbol{t})$. Then we use the next-token probabilities of tokens whose indices belong in $\boldsymbol{L}$ to build a probabilistic program, which we

can then use to further sample from repeatedly. Algorithm 2 captures our approach, where `token_analysis` and `compile` are the two subroutines for selecting tokens and producing **the concrete** probabilistic programs, respectively.

Algorithm 2 Compute Efficient Inference Strategy: Probabilistic Programs of Thought

Input Prompt t, sequence length n, number of LLM generations k, number of samples using one LLM generation m

```
samples ← []
probs ← {}                                          ▷ An empty dictionary
for j ∈ {1,2,...,k}                                 } Repeat k times
    for i ∈ {1,2,...,n}
        x_i ← sample(P_M(. | x_<i, t))
        probs[i] ← P_M(. | x_<i, t)
    prog ← parse(x_1, x_2, ... x_n)
    L ← token_analysis(prog)
    Prog ← compile(prog, probs[L])
    for i' ∈ {1,2,...,m}                            } Sample from probabilistic program m times
        prog_new ← sample(Prog)
        exc ← execute(prog_new)
        samples ← samples + [(prog_new, exc)]
return samples
```

Note that unlike Algorithm 1, Algorithm 2 generates $m \cdot k$ samples using the same number of LLM generations (i.e. k). These $m \cdot k$ samples will not be independent: each group of m samples shares all tokens at indices outside $\boldsymbol{L}$. Nevertheless, program execution is highly sensitive to local token changes. Substituting a single digit or operator can produce a completely different return value, so even samples that differ at only a few positions can yield distinct execution results and can therefore substantially improve the coverage with respect to P_{code}.

Because Algorithm 2 resamples only the tokens at positions in $\boldsymbol{L}$ without resampling the succeeding tokens, the resulting samples are not distributed according to P_{code} without additional independence assumptions. Given the assumption that each resampled token at a position in $\boldsymbol{L}$ is independent of the succeeding tokens in the LLM sequence, we show that the empirical distribution of samples from Algorithm 2 converges almost surely to P_{code}. We formally state the assumption below and include the convergence theorem and its proof in Appendix A.

Assumption 1 (Independence of Resampled Tokens). *Let $\boldsymbol{L} \subseteq \{1, 2, \ldots n\}$ be a subset of token indices to be treated as random variables. For each token index $j \in \boldsymbol{L}$, we assume that the succeeding tokens are independent of the j-th token given the preceding tokens. Formally,*

$$\forall j \in \boldsymbol{L}, j < i < n; P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i}, \boldsymbol{t}) = P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i \setminus j}, \boldsymbol{t})$$

where $\boldsymbol{X}_{<k \setminus j}$ refers to the sequence of tokens before k excluding the j-th token.

Ahmed et al. (2024) employs similar independence assumptions to incorporate symbolic constraints in autoregressive models. In our experiments, we show that this assumption is sufficiently robust to produce practically useful samples.

3.3 Compiling PPoT

So far we have abstracted away how to extract a probabilistic program to represent (part of) the LLM's distribution over deterministic programs. Our goal is to select program components (e.g. constants, digits, operators, function and variable names) and turn these components into random variables whose support is over possible syntactically valid values those variables could take. As an example, we may convert a `+` sign in a deterministic program into a random variable over the possible arithmetic operators {`+`,`-`,`*`,`/`,`//`,`**`}. This

	GSM8k			CRUXEval			Plot2Code	
	Qwen2.5 Coder			Qwen2.5 Coder			Qwen2.5 VL	
$k \cdot (\text{LLM} + m \cdot \text{PP})$	0.5B	3B	7B	0.5B	3B	7B	3B	7B
$1 \cdot (\text{LLM} + 0 \cdot \text{PP})$	24.94	72.63	82.71	30.87	40.5	57.12	39.44	36.70
$1 \cdot (\text{LLM} + 5 \cdot \text{PP})$	**28.51**	**76.19**	**84.76**	**37.37**	**45.87**	**60.62**	**41.89**	**42.48**
$5 \cdot (\text{LLM} + 0 \cdot \text{PP})$	42.53	86.20	91.20	46.13	59.13	73.37	63.86	69.92
$5 \cdot (\text{LLM} + 5 \cdot \text{PP})$	**49.13**	**89.08**	**93.02**	**53.13**	**63.87**	**76.87**	**64.84**	**71.95**
Metric	Accuracy						Text Match	

Table 1: **PPoT consistently and significantly boosts performance.** Each entry shows the score for each dataset (according to that dataset's metric) when using k LLM samples and m PPoT samples for each LLM generated program.

new probabilistic program now encodes a distribution over several different deterministic programs.

Although PPoT is agnostic to the choice of which program component is turned into a random variable, in practice the choices can be limited. If the program component under consideration spans multiple tokens, it is computationally #P-hard to compute the parameters of the categorical distribution associated with the corresponding random variable. We provide details in Appendix C.

For the current work, we limit our choices to the tractable case where the program components are contained within a single token in the vocabulary $\mathcal{V}$ of the language model. For example, digits and arithmetic operators are often contained in $\mathcal{V}$ as single tokens. This reduces Equation (5) in Appendix C to simply setting the distribution of a random variable C as the next-token distribution

$$P(C = c|t) := \begin{cases} \frac{P_{\mathcal{M}}(X_i = c|\boldsymbol{X}_{<i}, \boldsymbol{t})}{\sum_{c' \in \text{Val}(C)} P_{\mathcal{M}}(X_i = c'|\boldsymbol{X}_{<i}, \boldsymbol{t})} & \text{if } c \in \text{Val}(C), \\ 0 & \text{otherwise;} \end{cases} \tag{4}$$

where here we use $\text{Val}(C)$ to mean the support of C, and i is the position of C in the sequence of tokens.

As we shall see in the next section, merely incorporating digits, comparison, arithmetic and assignment operators as random variables in PPoT already substantially increases performance in code and structured output generation.

4 Evaluating PPoT...

We empirically validate PPoT on Qwen2.5 Coder (Hui et al., 2024) and Qwen2.5 VL (Bai et al., 2025) across model sizes and three different datasets: GSM8k (Cobbe et al., 2021b), Plot2Code (Wu et al., 2024) and CRUXEval (Gu et al., 2024b). In all experiments, the model is prompted to output the desired result (prompts in Appendix B) and we apply Best-of-n (Cobbe et al., 2021b): all N candidates are executed and scored by a verifier (Appendix G). For each of k LLM samples, we generate m additional samples from PPoT, for a total of $k \cdot (1+m)$ candidates per problem. Since PPoT samples require no GPU forward passes, only the k LLM samples contribute to computational cost; the remaining $k \cdot m$ samples are computationally cheap (see Section 4.2). We evaluate using the Best-of-n framework (Chen et al., 2021) with $N = k + k \cdot m$. Results are summarized in Table 1; qualitative examples can be found in Appendix F.

4.1 ... for Performance

Mathematical Reasoning (GSM8k). We evaluate PPoT + Qwen2.5 Coder (0.5B, 3B, 7B) on GSM8k, reporting accuracy (see Figure 1). We adopt the tractable compilation from Section 3.3, choosing digits, comparison, arithmetic and assignment operators as random variables, and

use `observe(.)` to sample conditioned on not reproducing the LLM sample (Appendices C and D). With only m=5 PPoT samples, accuracy improves by 2–7% without any additional GPU compute. Figure 3 shows accuracy trends as k increases: the dashed line shows that PPoT achieves the performance of 20 LLM samples using only 8.

Code Generation (Plot2Code). We evaluate PPoT + Qwen2.5 VL (3B, 7B) on the `matplotlib` split of Plot2Code, reporting mean text match score (Wu et al., 2024). We apply the same tractable compilation and conditioned sampling as in GSM8k. With m=5, performance improves by 0.5–6% across model sizes without additional GPU compute. Equivalent accuracy-vs-k graphs can be found in Appendix E. Figure 4 shows a case where PPoT is able to generate a very similar plot to the ground-truth while the LLM produced a runtime error.

Structured Output Generation (CRUXEval). PPoT applies beyond code generation to any structured output that can be verified. We evaluate PPoT + Qwen2.5 Coder (0.5B, 3B, 7B) on CRUXEval, a program inversion task (Pierce et al., 2026; Charguéraud, 2026): given a Python function and its output, generate an input that produces that output. Unlike GSM8k and Plot2Code, the output here is a structured Python object (e.g. a list or string) rather than a program. We exploit this structure via *subset sampling*: for each LLM-generated sequence, we resample a subset of tokens using the next-token probabilities. For example, given `[1,2,3]`, the comma after `2` could have been a closing bracket `]`, yielding `[1,2]`—a valid alternative recovered from the full joint distribution. This yields many valid samples from a single LLM generation, further diversified by conditioning on not reproducing the original (Algorithms 3 and 4 in Appendix H). PPoT boosts accuracy by 3–7% across model sizes without additional GPU compute (Figure 7 in Appendix E).

4.2 ... for Efficiency

PPoT compilation and sampling require no GPU forward passes. In practice, the wall-clock runtime is dominated by LLM inference: Figure 5 shows that the runtime curves for m=0 and m=20 are nearly indistinguishable, and the same holds for Plot2Code and CRUXEval (Figure 10 and Figure 8). The accuracy gains from PPoT therefore come at near-zero computational cost.

To measure the computational efficiency of PPoT, we must capture quality of PPoT samples relative to LLM samples. For this analysis, we fit a scaling law $1 - \text{acc}(k; m) = a_m \cdot k^{-b_m}$ to each accuracy curve in Figure 3 and use the LLM-only fit (m=0) to compute how many additional LLM samples would be needed to match any LLM+PPoT accuracy without PPoT (see Appendix I for details). Figure 6 plots this quantity across model sizes: for 20 LLM samples (k=20), even getting a single PPoT sample ($m = 1$) per LLM sample provides accuracy equivalent to ∼8 additional LLM samples, while 20 PPoT samples (m=20) is

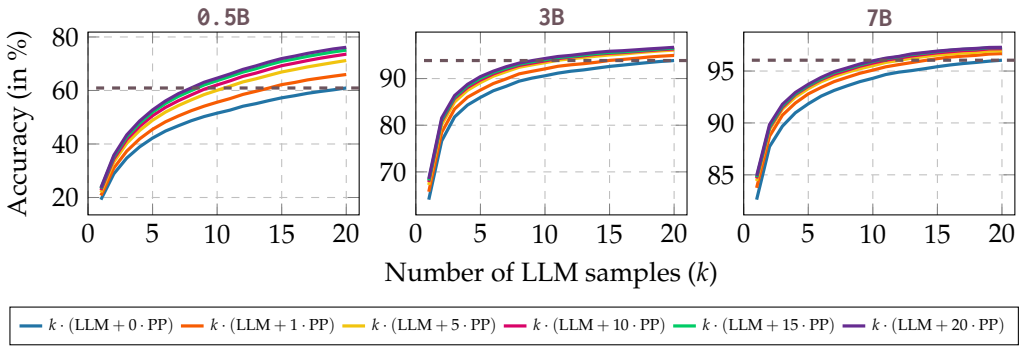


Figure 3: **PPoT achieves similar or higher performance with fewer LLM samples.** Each curve shows the pass@$(k + k \cdot m)$ accuracy of k LLM samples on GSM8k, each of which having been boosted by m samples from PPoT, where $m \in \{0, 1, 5, 10, 15, 20\}$. The gray dashed line highlights how many fewer (from the 20) LLM samples we can get away with if we use m many PPoT samples.

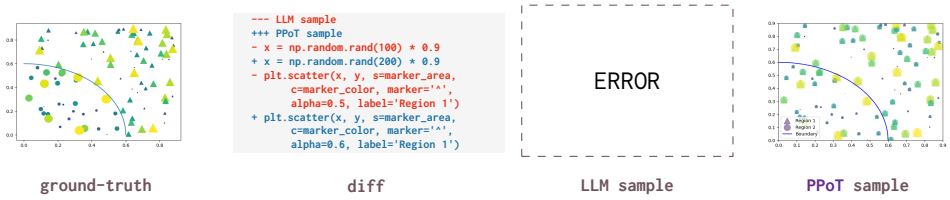


Figure 4: **PPoT enables cheap samples that correct the LLM generated program.** From left to right, we show the ground-truth plot, followed by a `diff` between the LLM and PPoT samples, the LLM sample rendering and then the resulting PPoT sample. The LLM generated code contains a runtime error and so produced no graph.

equivalent to $\sim$39 additional LLM samples. Moreover, this surplus grows with k, meaning PPoT becomes increasingly valuable as one scales up LLM sampling.

5 Related Work

Prior work has used language models for statistical modeling by using LLMs to generate programs in specialized PPLs such as Church (Wong et al., 2023), Stan (Domke, 2025), Problog (Cai et al., 2025), and PyMC (Kanda et al., 2026). Separately, the probabilistic programming perspective of separating modeling from inference has been applied to constrained decoding and alignment, using SMC (Loula et al., 2025; Lew et al., 2023) and MCMC (Faria & Smith, 2025) to guide LLM sampling (Korbak et al., 2022). Other work combines LLM outputs with token probabilities for structured reasoning (Dohan et al., 2022; Wan et al., 2025; Kamali & Kordjamshidi, 2025; Feng et al., 2025; You et al., 2025; Cheng et al., 2026). Our work differs from all three lines: we interpret LLM-generated *Python* code directly as a probabilistic program, avoiding specialized PPLs and treating the generated program as a tractable representation of language model distribution. A fuller discussion of related work is in Appendix J.

6 Conclusion

In this paper, we proposed *probabilistic programs of thought* (PPoT), a novel test-time framework for efficient language model code generation. We described how an LLM prompted to generate code can be interpreted as a probabilistic program, and how one can approximate this through PPoT. From this approximation, one can then tractably perform probabilistic reasoning. Our approach uses this probabilistic program to sample alternate generations and consequently reduces the number of LLM calls. We envision using these probabilistic

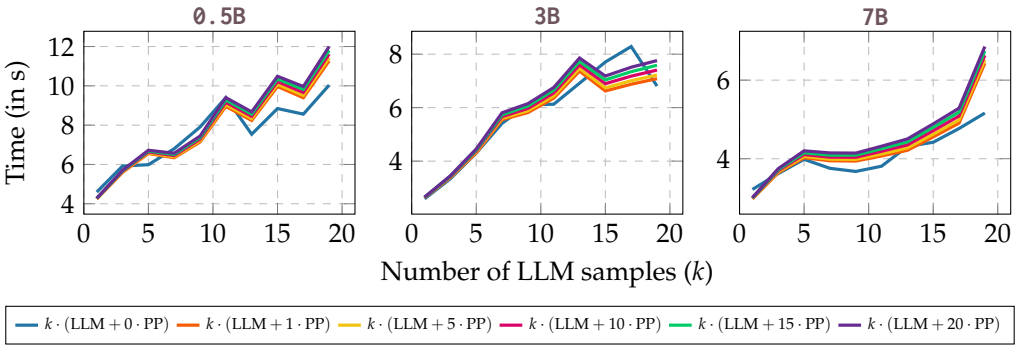


Figure 5: **PPoT compilation and sampling time is computationally cheap.** Each curve shows the runtime of sampling k LLM samples, compiling and then sampling m programs from PPoT on GSM8k, where $m \in \{0, 1, 5, 10, 15, 20\}$. When $m = 0$, no compilation or sampling is done.

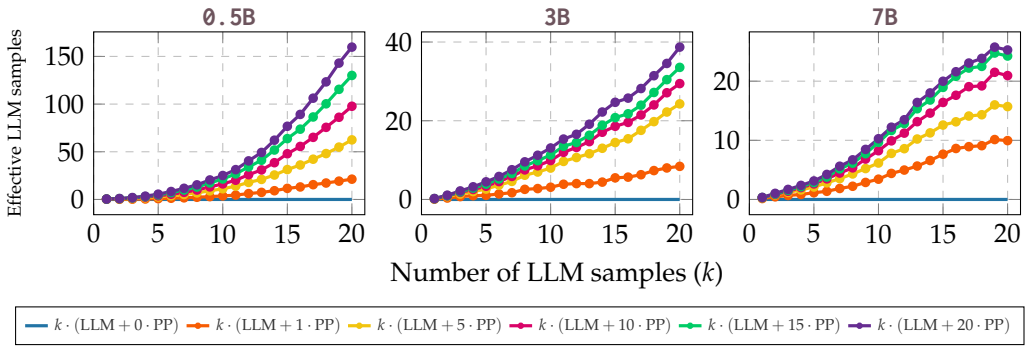


Figure 6: **Effective LLM samples from PPoT.** For each configuration of k LLM samples with $m \in \{0, 1, 5, 10, 15, 20\}$ PPoT samples each, we compute how many additional LLM-only samples would be needed to match the same accuracy using the LLM-only scaling law.

programs for other probabilistic queries to support more language model tasks. For example, programs can be conditioned or can be used for computing expectation while using fewer samples from the language models. We also foresee expanding the structures being interpreted as random variables beyond integers, arithmetic operators to sub-expressions and control flow structure to enable richer probabilistic programs. Overall, we believe that this work opens a new direction of research that uses probabilistic programs as abstractions of language model generations.

Acknowledgements

The authors would like to thank Kareem Ahmed, Oliver Broadrick, Zilei Zoe Shao, Benjie Wang and Zhe Zeng for insightful technical discussions. This work was funded in part by the DARPA ANSR, CODORD, and SAFRON programs under awards FA8750-23-2-0004, HR00112590089, and HR00112530141, NSF grant IIS1943641, NSF grant CCF-2220891, and gifts from Adobe Research, Cisco Research, Qualcomm, and Amazon. Approved for public release; distribution is unlimited.

References

Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J. Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas Wiecki, and Robert Zinkov. PyMC: A modern and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science*, 9(e1516), 2023. doi: 10.7717/peerj-cs.1516.

Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. A pseudo-semantic loss for autoregressive models with logical constraints, 2024. URL https://arxiv.org/abs/2312.03905.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report, 2025. URL https://arxiv.org/abs/2502.13923.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):1946–1969, 2023.

Vladimir I. Bogachev. *Measure Theory*. Springer Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-34514-5.

Zhixi Cai, Fucai Ke, Simindokht Jahangard, Maria Garcia de la Banda, Reza Haffari, Peter J. Stuckey, and Hamid Rezatofighi. Naver: A neuro-symbolic compositional automaton for visual grounding with explicit logic reasoning, 2025. URL https://arxiv.org/abs/2502.00372.

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017. doi: 10.18637/jss.v076.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v076i01.

Arthur Charguéraud. *Separation Logic Foundations*, volume 6 of *Software Foundations*. Electronic textbook, 2026. Version 3.0, http://softwarefoundations.cis.upenn.edu.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023. URL https://arxiv.org/abs/2211.12588.

Junyan Cheng, Kyle Richardson, and Peter Chin. Analytica: Soft propositional reasoning for robust and scalable llm-driven analysis. In *The Fourteenth International Conference on Learning Representations*, 2026.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021a. URL https://arxiv.org/abs/2110.14168.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021b. URL https://arxiv.org/abs/2110.14168.

D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *IEEE Transactions on Signal Processing*, 50(3):736–746, 2002. doi: 10.1109/78.984773.

Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. Semantics of probabilistic programming: A gentle introduction. *Foundations of Probabilistic Programming*, pp. 1, 2020.

Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language model cascades, 2022. URL https://arxiv.org/abs/2207.10342.

Justin Domke. Large language bayes, 2025. URL https://arxiv.org/abs/2504.14025.

Gonçalo Faria and Noah A. Smith. Sample, don't search: Rethinking test-time alignment for language models, 2025. URL https://arxiv.org/abs/2504.03790.

Yu Feng, Ben Zhou, Weidong Lin, and Dan Roth. Bird: A trustworthy bayesian inference framework for large language models, 2025. URL https://arxiv.org/abs/2404.12494.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015. doi: 10.1017/S1471068414000076.

Renato Geh, Honghua Zhang, Kareem Ahmed, Benjie Wang, and Guy Van Den Broeck. Where is the signal in tokenization space? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3966–3979, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.230. URL https://aclanthology.org/2024.emnlp-main.230/.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, pp. 220–229, Arlington, Virginia, USA, 2008. AUAI Press. ISBN 0974903949.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Future of software engineering proceedings*, pp. 167–181. 2014.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution, 2024a. URL https://arxiv.org/abs/2401.03065.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution, 2024b. URL https://arxiv.org/abs/2401.03065.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14953–14962, 2023.

Theodore P. Hill and James Mann. Alternative empirical distributions based on weighted linear combinations of order statistics. *Stochastic Analysis and Applications*, 18(1):87–99, 2000. doi: 10.1080/07362990008809656. URL https://doi.org/10.1080/07362990008809656.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Sound abstraction and decomposition of probabilistic programs. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1999–2008. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/holtzen18a.html.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–31, 2020.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, January 2026. ISSN 1557-7392. doi: 10.1145/3747588. URL http://dx.doi.org/10.1145/3747588.

Danial Kamali and Parisa Kordjamshidi. Neptune: A neuro-pythonic framework for tunable compositional reasoning on vision-language, 2025. URL https://arxiv.org/abs/2509.25757.

Madhav Kanda, Shubham Ugare, and Sasa Misailovic. Refinestat: Efficient exploration for probabilistic program synthesis, 2026. URL https://arxiv.org/abs/2509.01082.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Jacek Karwowski, Younesse Kaddar, Zihuiwen Ye, Nikolay Malkin, and Sam Staton. Likelihood hacking in probabilistic program synthesis, 2026. URL https://arxiv.org/abs/2603.24126.

Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. Understanding probabilistic programs. In *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015, Proceedings*, pp. 15–32. Springer, 2015.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Tomasz Korbak, Ethan Perez, and Christopher L Buckley. Rl with kl penalties is better viewed as bayesian inference, 2022. URL https://arxiv.org/abs/2205.11275.

Alexander K. Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K. Mansinghka. Sequential monte carlo steering of large language models using probabilistic programs, 2023. URL https://arxiv.org/abs/2306.03081.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. Syntactic and semantic control of large language models via sequential monte carlo, 2025. URL https://arxiv.org/abs/2504.13139.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

Avi Pfeffer. *Practical Probabilistic Programming*. Manning Publications Co., USA, 1st edition, 2016. ISBN 1617292338.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2026. Version 7.0, http://softwarefoundations.cis.upenn.edu.

Kyle Richardson, Vivek Srikumar, and Ashish Sabharwal. Understanding the logic of direct preference alignment through logic. In *Forty-second International Conference on Machine Learning*, 2025.

Rylan Schaeffer, Joshua Kazdan, John Hughes, Jordan Juravsky, Sara Price, Aengus Lynch, Erik Jones, Robert Kirk, Azalia Mirhoseini, and Sanmi Koyejo. How do large language monkeys get their power (laws)? *arXiv preprint arXiv:2502.17578*, 2025.

Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. Dspy assertions: Computational constraints for self-refining language model pipelines. *arXiv preprint arXiv:2312.13382*, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Wentao Wan, Kaiyu Wu, Qingyang Ma, Nan Kang, Yunjie Chen, Liang Lin, and Keze Wang. Enhancing visual programming for visual reasoning via probabilistic graphs, 2025. URL https://arxiv.org/abs/2512.14257.

Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought, 2023. URL https://arxiv.org/abs/2306.12672.

Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. Plot2code: A comprehensive benchmark for evaluating multimodal large language models in code generation from scientific plots, 2024. URL https://arxiv.org/abs/2405.07990.

Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, pp. 5502–5511. PMLR, 2018.

Weiqiu You, Anton Xue, Shreya Havaldar, Delip Rao, Helen Jin, Chris Callison-Burch, and Eric Wong. Probabilistic soundness guarantees in llm reasoning chains, 2025. URL https://arxiv.org/abs/2507.12948.

A Soundness of Algorithm 2

In Section 3.2, we treated the token corresponding to `0` as the random variable X which we resample in the probabilistic program. However, given the autoregressive nature of the language model distribution, resampling X requires resampling the succeeding tokens in the generated program. Since Algorithm 2 does not resample these succeeding tokens, the distribution of the samples obtained from Algorithm 2 will not be the same as P_{code}. We state the following independence assumption between the resampled tokens and the succeeding tokens as a sufficient condition for Algorithm 2 to be sound. This assumption states that all tokens succeeding the j-th token are independent of the j-th token given the tokens preceding it.

Assumption 2 (Independence of Resampled Tokens). *Let $\boldsymbol{L} \subseteq \{1, 2, \dots n\}$ be a subset of token indices to be treated as random variables. For each token index $j \in \boldsymbol{L}$, we assume that the succeeding tokens are independent of the j-th token given the preceding tokens. Formally,*

$$\forall j \in \boldsymbol{L}, j < i < n; P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i}, \boldsymbol{t}) = P_{\mathcal{M}}(X_i \mid \boldsymbol{X}_{<i\setminus j}, \boldsymbol{t})$$

where $\boldsymbol{X}_{<k\setminus j}$ refers to the sequence of tokens before k excluding the j-th token.

In fact, with the above independence assumption, Algorithm 2 is sound with respect to the code-generating probabilistic program P_{code} stated by the following theorem and the proof.

Theorem 1 (Soundness). *Let $\boldsymbol{s}_1, \boldsymbol{s}_2, \dots \boldsymbol{s}_k$ be the samples obtained from Algorithm 2. Then, under Assumption 2, independence of resampled tokens, the empirical distribution induced by the samples converges almost surely to P_{code} i.e., (where $\delta_{\boldsymbol{s}_i}$ is the Dirac delta distribution centered at $\boldsymbol{s}_i$):*

$$\sum_{i=1}^{k} \frac{1}{k} \delta_{\boldsymbol{s}_i} \xrightarrow[k\to\infty]{a.s.} P_{code}.$$

Proof. Let $\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{s}_3, \dots \boldsymbol{s}_k$ be the samples obtained from Algorithm 2. Since all samples are equally weighted, the empirical distribution (Hill & Mann, 2000) induced by these samples is $\sum_{i=1}^{k} \frac{1}{k} \delta_{\boldsymbol{s}_i}$. Here $\delta_{\boldsymbol{s}_i}$ is the Dirac delta distribution centered at $\boldsymbol{s}_i$, which assigns probability 1 to the point $\boldsymbol{s}_i$ and 0 to all other points.

We first show that each sample in $\boldsymbol{s}_1, \boldsymbol{s}_2, \dots, \boldsymbol{s}_k$ is drawn from the distribution P_{code}. Let $\boldsymbol{s}$ be a sample from Algorithm 2 and it consists of the program π made of tokens $(x_1, x_2, \dots x_n)$

and the result r. Let $\boldsymbol{L}$ be the set of tokens used to make the probabilistic program on Line 8–9 of Algorithm 2. Then the probability of sampling $\boldsymbol{s}$ can be expressed as

$$P(\boldsymbol{s}) = \prod_{i \notin \boldsymbol{L}} P_{\mathcal{M}}(x_i \mid \boldsymbol{x}_{<i}, \boldsymbol{t}) \prod_{j \in \boldsymbol{L}} P_{\mathcal{M}}(x_j \mid \boldsymbol{x}_{<j}, \boldsymbol{t}) = \prod_{i=1}^{n} P_{\mathcal{M}}(x_i \mid \boldsymbol{x}_{<i}, \boldsymbol{t}) \qquad \text{(Assumption 2)}$$

$$= \prod_{i=1}^{n} P_{\mathcal{M}}(x_i \mid \boldsymbol{x}_{<i}, \boldsymbol{t}) \cdot [\![\pi \doteq \texttt{parse}(x_1, x_2, \ldots, x_n)]\!] \cdot [\![r \doteq \texttt{execute}(\pi)]\!] = P_{code}(\boldsymbol{s})$$

Now that each sample $\boldsymbol{s}_i$ is drawn from the distribution P_{code}, we can show that for any arbitrary generation $\boldsymbol{g}$, the probability of $\boldsymbol{g}$ under the empirical distribution converges to $P_{code}(\boldsymbol{g})$.

$$\lim_{k \to \infty} \left(\sum_{i=1}^{k} \frac{1}{k} \delta_{\boldsymbol{s}_i} \right) (\boldsymbol{g}) = \lim_{k \to \infty} \sum_{i=1}^{k} \frac{1}{k} \delta_{\boldsymbol{s}_i}(\boldsymbol{g}) = \lim_{k \to \infty} \frac{1}{k} \sum_{i=1}^{k} [\![\boldsymbol{s}_i = \boldsymbol{g}]\!]$$

$$= \lim_{k \to \infty} \frac{1}{k} k \cdot P_{code}(\boldsymbol{g}) = P_{code}(\boldsymbol{g})$$

□

One can also view Algorithm 2 as a special case of Sequential Monte Carlo (Crisan & Doucet, 2002) with $\frac{k}{m}$ particles where we uniformly resample them into M samples on Line 10–14 of Algorithm 2. Under this perspective, the convergence of the empirical distribution to P_{code} can also be shown using the convergence results for Sequential Monte Carlo.

B Prompts

We use the standard system prompt for **Qwen2.5Coder** as recommended in the model's HuggingFace documentation:

```
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.
```

The user prompt used in **GSM8k** is as follows

```
Generate a Python function 'compute_answer' with no arguments that computes the
needed calculations and returns a number as the answer to the problem below. There
should be no comments in the code. Only generate the Python function
'compute_answer', with no explanations and no user input. The function should show
intermediate computations in the program but without any comments.\n\nProblem:
{question}
```

where `{question}` is replaced with the **GSM8k** problem statement.

The user prompt used in **Plot2Code** is as follows:

```
Please generate Python matplotlib code to create a plot that looks like the given
image. The code should be surrounded by ```python and ```.
```

The user prompt used in **CRUXEval** is as follows:

```
You will be given a function f and an output in the form f(??) == output. Find any
input such that executing f on the input leads to the given output. There may be
multiple answers, but you should only output one. In [ANSWER] and [/ANSWER] tags,
complete the assertion with one such input that will produce the output when
executing the function.

[PYTHON]
def f(my_list):
    count = 0
```

```
    for i in my_list:
        if len(i) % 2 == 0:
            count += 1
    return count
assert f(??) == 3
[/PYTHON]
[ANSWER]
assert f(["mq", "px", "zy"]) == 3
[/ANSWER]

[PYTHON]
def f(s1, s2):
    return s1 + s2
assert f(??) == "banana"
[/PYTHON]
[ANSWER]
assert f("ba", "nana") == "banana"
[/ANSWER]

[PYTHON]
{code}
assert f(??) == {output}
[/PYTHON]
[ANSWER]
```

C Compiling random variables in PPoT

By Assumption 2, any program component C (e.g. constants, digits, operators, function and variable names, etc.) at position i in the sequence of tokens with possible values $\text{Val}(C)$ can be made into a random variable by setting its distribution as the marginalized distribution (over tokens) for each $c \in \text{Val}(C)$ and then appropriately renormalizing according to its support and the language model's vocabulary $\mathcal{V}$

$$P(C = c|t) := \begin{cases} \frac{\sum_{\boldsymbol{x} \models c, \forall \boldsymbol{x} \in \mathcal{V}^*} P_{\mathcal{M}}(\boldsymbol{X}=\boldsymbol{x}|X_{<i},t)}{\sum_{c' \in \text{Val}(C)} \sum_{\boldsymbol{x}' \models c', \forall \boldsymbol{x}' \in \mathcal{V}^*} P_{\mathcal{M}}(\boldsymbol{X}=\boldsymbol{x}'|X_{<i},t)} & \text{if } c \in \text{Val}(C), \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Here we use $\boldsymbol{x} \models c$ to denote a sequence of tokens $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ that detokenizes into c, i.e. $x_1 \circ x_2 \circ \cdots \circ x_m = c$, with $\circ$ denoting string concatenation Unfortunately, computing these marginals is known to be #P-hard (Geh et al., 2024), and so it is infeasible to retrieve $P(C|t)$.

Assuming random variables are single token, we can easily set the support of a random variable according to the possible syntactically equivalent values that program component could take. In our implementation, any time one of the tokens below is detected in the program to be compiled, we turn that token into a random variable, set its distribution to the next-token distribution of that token, and then restrict it to the support according to its syntactic category below, renormalizing appropriately afterwards.

Syntactic category	Support
DIGITS	$\text{Val}(C_{\text{digits}}) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
COMPARISON	$\text{Val}(C_{\text{cop}}) = \{<=, >=, ==, >, <, !=, \llcorner<=, \llcorner>=, \llcorner==, \llcorner>, \llcorner<, \llcorner!=\}$
ARITHMETIC	$\text{Val}(C_{\text{bop}}) = \{+, -, *, /, //, **, \llcorner+, \llcorner-, \llcorner*, \llcorner/, \llcorner//, \llcorner**, (+, (-\}$
ASSIGNMENT	$\text{Val}(C_{\text{aop}}) = \{+=, -=, *=, /=, //=, \llcorner+=, \llcorner-=, \llcorner*=, \llcorner/=, \llcorner//=\}$

Here, we use $\llcorner$ to denote a whitespace (as part of the token itself).

D Sampling from PPoT

In order to retrieve more diverse generations, we sample conditioned on not sampling the LLM sample PPoT was compiled from. Below we show a more general formulation where we sample conditioned on not sampling a set of previously seen examples. This can be done tractably because of Assumption 2: a fully factorized distribution admits trivial sampling without replacement. We shall now describe how this sampling without replacement is done.

Let $\boldsymbol{C} = (C_1, C_2, \ldots, C_m)$ be the set of all random variables in a probabilistic program Δ. Denote by P_Δ the probability mass function of the probability distribution induced by Δ. Our goal is to sample a deterministic program ρ that is different from the previous k samples $\boldsymbol{\rho} = \{\rho_1, \rho_2, \rho_3, \ldots, \rho_k\}$. For completeness sake, we condition on the LLM prompt t that generated the sketch of the probabilistic program.

$$\rho \sim P_\Delta\left(\boldsymbol{C} | \rho \notin \boldsymbol{\rho}, \boldsymbol{t}\right).$$

In other words, let $\phi = \bigwedge_{i=1}^{k} (\rho \neq \rho_i)$ be the constraint of sampling without replacement, we wish to sample as follows

$$\rho \sim P_\Delta\left(\boldsymbol{C} | \phi, \boldsymbol{t}\right).$$

We can rewrite ϕ in terms of the random variables, where here we use lower case to denote the valuations of the corresponding random variables

$$\phi = \neg \left(\bigvee_{i=1}^{k} \left[\left(c_1^{(\rho)} = c_1^{(\rho_i)} \right) \wedge \left(c_2^{(\rho)} = c_2^{(\rho_i)} \right) \wedge \cdots \wedge \left(c_m^{(\rho)} = c_m^{(\rho_i)} \right) \right] \right).$$

The probability of sampling $\boldsymbol{C}$ then becomes

$$P_\Delta(C_1, C_2, \ldots, C_m | \phi, \boldsymbol{t}) = \prod_{i=1}^{m} P_\Delta(C_i | \phi, \boldsymbol{t}) = \prod_{i=1}^{m} \frac{P_\Delta(C_i, \phi | \boldsymbol{t})}{P_\Delta(\phi | \boldsymbol{t})} = \prod_{i=1}^{m} \frac{P_\Delta(C_i | \boldsymbol{t}) - P_\Delta(C_i, \neg\phi | \boldsymbol{t})}{1 - P_\Delta(\neg\phi | \boldsymbol{t})}. \tag{6}$$

Note that

$$P_\Delta(\neg\phi | \boldsymbol{t}) = \sum_{i=1}^{k} P_\Delta\left(c_1^{(\rho_i)}, c_2^{(\rho_i)}, \ldots, c_m^{(\rho_i)} \middle| \boldsymbol{t} \right) = \sum_{i=1}^{k} \prod_{j=1}^{m} P_\Delta\left(c_j^{(\rho_i)} \middle| \boldsymbol{t} \right)$$

can be computed efficiently, since all disjuncts are mutually exclusive and independent in $\neg\phi$ and Assumption 2 allows us to fully factorize the distribution. The numerator can be computed tractably as $P_\Delta(C_i | t)$ is simply the next-token distribution (tractably renormalized on its support) and

$$P_\Delta(C_i, \neg\phi | \boldsymbol{t}) = \begin{cases} P_\Delta(\neg\phi | \boldsymbol{t}) & \text{if } \neg\phi_i \models C_i, \\ 0 & \text{otherwise;} \end{cases}$$

here we use the standard propositional logic semantics for $\models$.

Intuitively, Equation (6) states that each random variable is sampled according to its distribution but penalized by the probability mass of past seen samples. We then reweight appropriately according to the remaining mass that is satisfiable with the constraint.

E Additional Experiment Details

All the experiments were run on a server with 32 CPU cores and 252 GB RAM, equipped with 2 NVIDIA RTX A6000 GPUs each with 48 GB memory.

Section 4 describes the evaluation of PPoT on the tasks of code generation, mathematical reasoning and structured output generation. Figure 3 and Figure 5 show trends in accuracy and time overhead for GSM8k across the number of LLM samples. Here, we include analogous plots for Plot2Code (Figure 9 and Figure 10) and CRUXEval (Figure 7 and Figure 8).

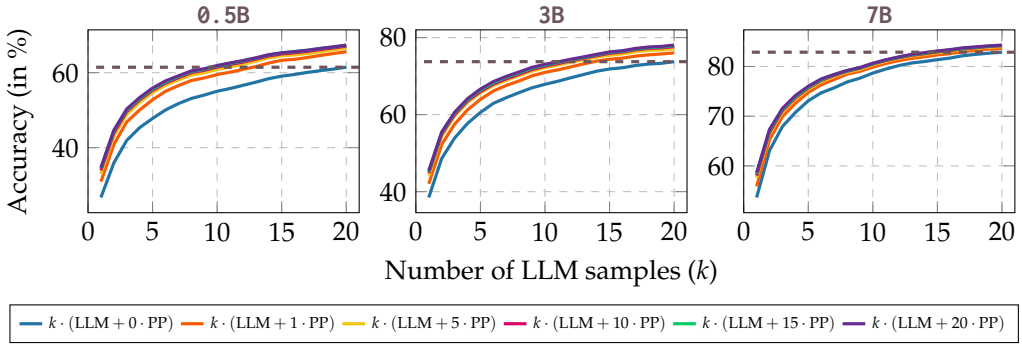


Figure 7: **PPoT achieves similar or higher performance with fewer LLM samples.** Each curve shows the pass@$(k + k \cdot m)$ accuracy of k LLM samples on CRUXEval, each of which having been boosted by m samples from PPoT, where $m \in \{0, 1, 5, 10, 15, 20\}$. The gray dashed line highlights how many fewer (from the 20) LLM samples we can get away with if we use m many PPoT samples.

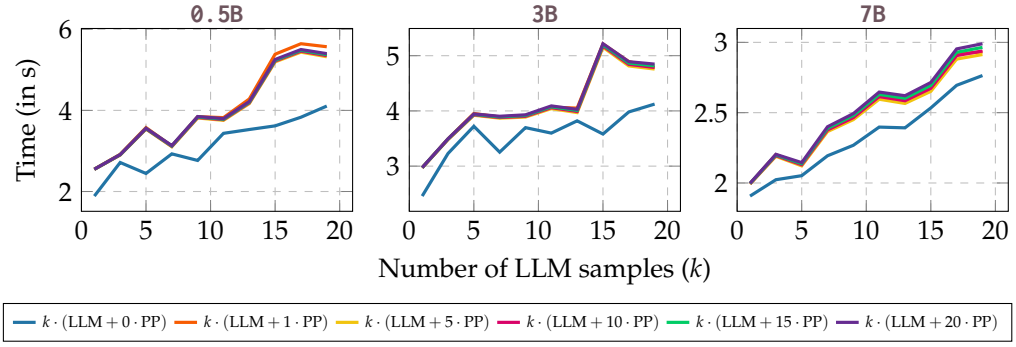


Figure 8: **PPoT compilation and sampling time is computationally cheap.** Each curve shows the runtime of sampling k LLM samples, compiling and then sampling m programs from PPoT on CRUXEval, where $m \in \{0, 1, 5, 10, 15, 20\}$. When $m = 0$, no compilation or sampling is done.

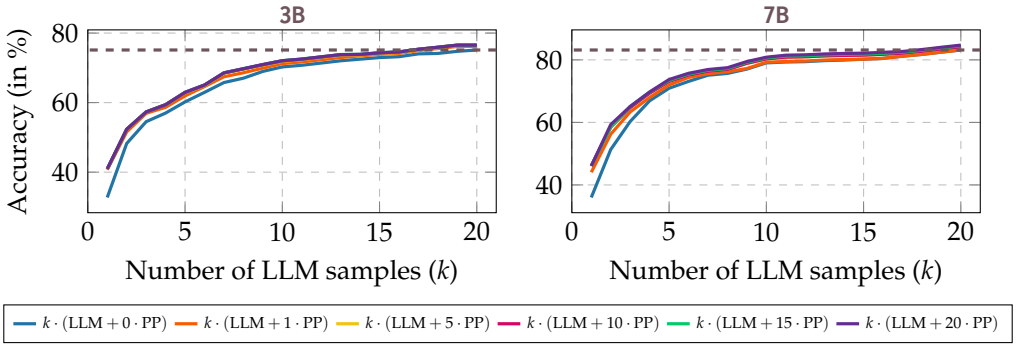


Figure 9: **PPoT achieves similar or higher performance with fewer LLM samples.** Each curve shows the pass@$(k + k \cdot m)$ accuracy of k LLM samples on Plot2Code, each of which having been boosted by m samples from PPoT, where $m \in \{0, 1, 5, 10, 15, 20\}$. The gray dashed line highlights how many fewer (from the 20) LLM samples we can get away with if we use m many PPoT samples.

F Qualitative examples of PPoT

In this section we show some examples where PPoT performs better compared to the LLM samples. We start with some examples in Plot2Code. In the examples below, we show the

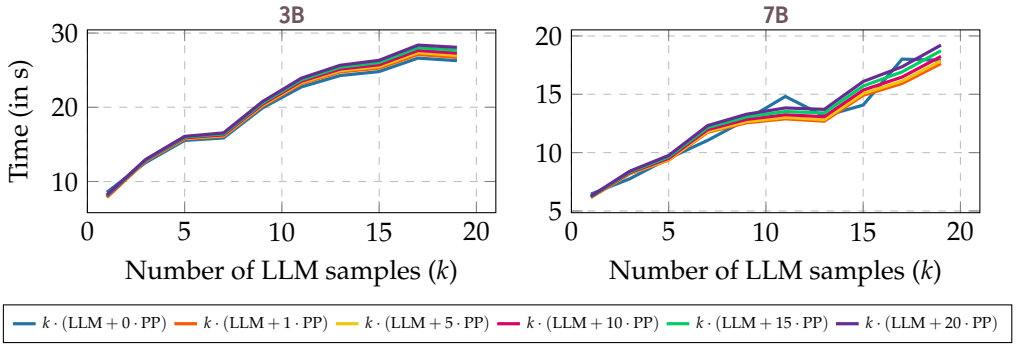
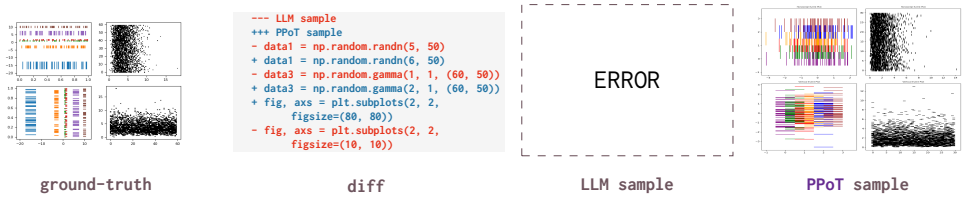


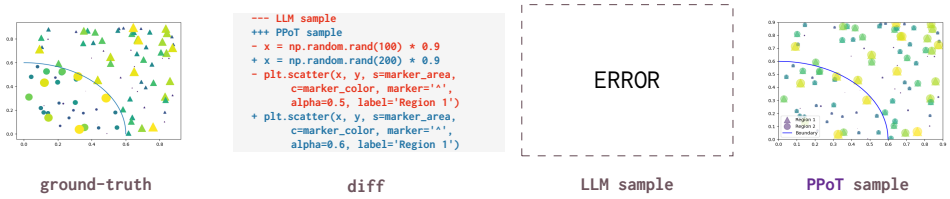
Figure 10: **PPoT compilation and sampling time is computationally cheap.** Each curve shows the runtime of sampling k LLM samples, compiling and then sampling m programs from PPoT on Plot2Code, where $m \in \{0, 1, 5, 10, 15, 20\}$. When $m = 0$, no compilation or sampling is done.

ground-truth plot, a `diff` comparing the LLM sample to the PPoT sample, and the resulting plots from these two.

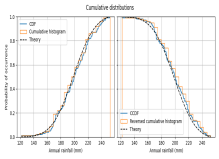
The first two examples show cases where the LLM sample was unable to render a graph due to runtime errors. In the first, the error lies in the dimension of the vectors `data1` and `data3` that did not match with the subsequent computation. The uncertainty of the model generated a `5` instead of a `6` in the case of `data1`, and a `1` instead of a `2` in `data3`. By accounting for this uncertainty through more samples and verifying them against a score, PPoT is able to ultimately produce better quality samples.



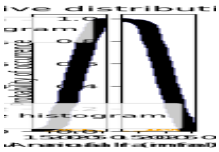
The second example is similar in spirit: here, `y` has length 100, yet the LLM sample defines `x` to be of size 200. The function `plt.scatter` expects both `x` and `y` to have the same dimensions. PPoT fixes this by setting the size of `x` to match `y`'s.

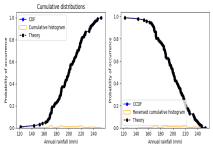


The issue with the next example is one of figure size: the LLM generated code produces a figure with width too small, which the PPoT sample then corrects according to the uncertainty of the approximated next-token distribution.



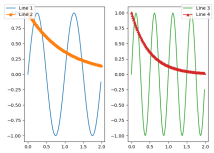
```
--- LLM sample
+++ PPoT sample
  # Plot the CDF
- plt.figure(figsize=(1, 4))
+ plt.figure(figsize=(9, 4))
  plt.subplot(1, 2, 1)
  plt.plot(x, y, 'o-', color='blue',
           label='CDF')
  plt.hist(x, bins=25, density=True,
           histtype='step', color='orange',
           label='Cumulative histogram')
```



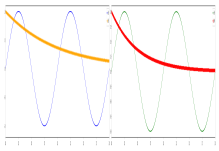


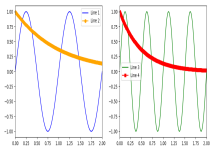
ground-truth diff LLM sample PPoT sample

The next example is similar to the previous one, in that the figure width is increased in the PPoT sample. More interesting, however, is the uncertainty in the definition of `y3`. By increasing the frequency from $f(x) = \sin(2 \cdot \pi \cdot x)$ to $f(x) = \sin(4 \cdot \pi \cdot x)$, the plot in PPoT more resembles the ground-truth.



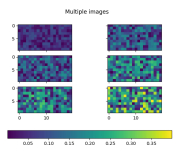
```
--- LLM sample
+++ PPoT sample
- y3 = np.sin(2 * np.pi * x)
+ y3 = np.sin(4 * np.pi * x)
- fig, (ax1, ax2) = plt.subplots(1, 2,
      figsize=(80, 4))
+ fig, (ax1, ax2) = plt.subplots(1, 2,
      figsize=(10, 4))
- ax1.set_ylim(-1.2, 1.1)
+ ax1.set_ylim(-1.1, 1.1)
```



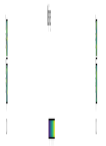


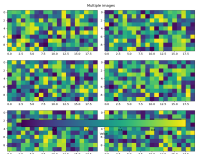
ground-truth diff LLM sample PPoT sample

Similarly to the previous examples, the next example shows that a simple missampling in the dimensions of `plt.subplots` can cause the plot to fail to produce better quality renders.



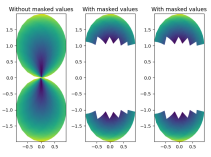
```
--- LLM sample
+++ PPoT sample
  # Create a figure with a grid of
      subplots
- fig, axs = plt.subplots(2, 2,
      figsize=(80, 1))
+ fig, axs = plt.subplots(3, 2,
      figsize=(10, 8))
  for ax, d in zip(axs.flat, data):
      im = ax.imshow(d,
               vmin=np.min(data),
               vmax=np.max(data))
```





ground-truth diff LLM sample PPoT sample

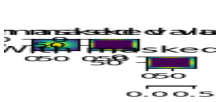
Again, the example below shows that a off-by-one mistake can produce unintelligible graphs that could be easily fixed with PPoT.

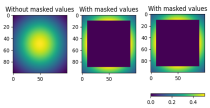


```
--- LLM sample
+++ PPoT sample
  Z = np.exp(-X**2 - Y**2)

  # Create a figure with 3 subplots
- fig, (ax0, ax1, ax2) = plt.subplots(1,
      3, figsize=(1, 3))
+ fig, (ax0, ax1, ax2) = plt.subplots(1,
      3, figsize=(6, 4))

  # Plot the 2D Gaussian distribution
      without masked values
```





ground-truth diff LLM sample PPoT sample

Next, we show some examples from GSM8K. In the examples below, we show the question from GSM8K, the ground truth answer and the diff between the LLM generation and PPoT sample. The first two examples here demonstrate the case where the language model did not sample the appropriate digit to solve the problem. However, it had significant probability mass on the correct digit that it was captured in the distribution of the probabilistic program leading to sampling of the correct program.

Question 4: Every day, Wendi feeds each of her chickens three cups of mixed chicken feed, containing seeds, mealworms and vegetables to help keep them healthy. She gives the chickens their feed in three separate meals. In the morning, she gives her flock of chickens 15 cups of feed. In the afternoon, she gives her chickens another 25 cups of feed. How many cups of feed does she need to give her chickens in the final meal of the day if the size of Wendi's flock is 20 chickens?
Ground truth answer: 20.0

```
--- LLM sample
+++ PPoT sample
  def compute_answer():
-     total_feed_needed = 20 * 3 * 3
+     total_feed_needed = 20 * 3 * 1
      morning_feed = 15
      afternoon_feed = 25
      final_meal_feed = total_feed_needed - (morning_feed +
            afternoon_feed)
      return final_meal_feed
```

Question 389: David and Dasha went to the carnival, where there are 9 rides. Each ride costs 2 ride tickets at $2 per ticket. You can also buy a ride bracelet for $30 which gives you 9 rides. If David buys a ride bracelet and Dasha buys tickets, and they ride all 9 rides, how much money does David save?
Ground truth answer: 6.0

```
--- LLM sample
+++ PPoT sample
- def compute_answer():
      ride_cost_per_ticket = 2
-     tickets_per_ride = 1
+     tickets_per_ride = 2
      ride_bracelet_cost = 30
      total_rides = 9

      total_cost_without_bracelet = total_rides *
            (ride_cost_per_ticket * tickets_per_ride)
      total_cost_with_bracelet = ride_bracelet_cost

      savings = total_cost_without_bracelet -
            total_cost_with_bracelet
      return savings
```

The following two examples demonstrate the case where **PPoT** samples an alternate arithmetic operator in the LLM program which leads to a correct program.

Question 92: Emily has 4 kids named Amy, Jackson, Corey, and James. Amy is 5 years older than Jackson and 2 years younger than Corey. If James is 10 and is 1 year younger than Corey, how old is Jackson?
Ground truth answer: 4.0

```
--- LLM sample
+++ PPoT sample
  def compute_answer():
      corey_age = 10 + 1
-     amy_age = corey_age + 2
+     amy_age = corey_age - 2
      jackson_age = amy_age - 5
      return jackson_age
```

Question 589: If Martin eats Cheerios every day for breakfast, he'll lose 1.25 pounds/week. If he eats donuts every day for breakfast, he'll gain 1.75 pounds/week. What will be the difference in his weight at the end of 5 weeks between the two breakfast options?
Ground truth answer: 15.0

```
--- LLM sample
+++ PPoT sample
  def compute_answer():
      weight_loss_per_week = 1.25
      weight_gain_per_week = 1.75
      number_of_weeks = 5
      total_weight_loss = weight_loss_per_week *
            number_of_weeks
      total_weight_gain = weight_gain_per_week *
            number_of_weeks
-     difference_in_weight = total_weight_gain -
       total_weight_loss
+     difference_in_weight = total_weight_gain +
       total_weight_loss
      return difference_in_weight
```

The following two examples demonstrate the case where **PPoT** resamples multiple tokens in the LLM generated program to get the correct program.

Question 122: Sadie slept 8 hours on Monday. For the next two days, she slept 2 hours less, each, because she had to complete some assignments. If the rest of the week she slept 1 hour more than those two days, how many hours did she sleep in total throughout the week?
Ground truth answer: 48.0

```
--- LLM sample
+++ PPoT sample
  def compute_answer():
      # Monday
      monday_sleep = 8

      # Tuesday and Wednesday
      tuesday_sleep = 8 - 2
      wednesday_sleep = 8 - 2

      # Thursday, Friday, Saturday, and Sunday
-     other_days_sleep = (8 + 2) * 4
+     other_days_sleep = (8 - 1) * 4

      # Total sleep for the week
      total_sleep = monday_sleep + tuesday_sleep +
            wednesday_sleep + other_days_sleep
      return total_sleep
```

Question 301: A simple folding newspaper or tabloid can be made by folding a piece of paper vertically and unfolding. Then, say, page 1 is printed on the left back, page 2 is printed on the left front, and then, perhaps page 32 is printed on the right back, and page 31 is printed on the right front. How many pieces of paper would be used in a 32-page tabloid?
Ground truth answer: 8.0

```
--- LLM sample
+++ PPoT sample
  def compute_answer():
      pages = 32
-     sheets = pages / 2
+     sheets = pages // 4
      return sheets
```

Next, we show some examples from **CRUXEval**. In the examples below, we show the expected output and the Python function f. We also show LLM's failed predicted input to get the given output and the successful **PPoT** sample.

Example 7

Expected Output: `[]`

LLM's Predicted Input: `f([1, 2, 3, 4, 5])`

PPoT sample: `f([5])`

```
def f(list):
    original = list[:]
    while len(list) > 1:
        list.pop(len(list) - 1)
        for i in range(len(list)):
            list.pop(i)
    list = original[:]
    if list:
        list.pop(0)
    return list
```

Example 19

Expected Output: `''`

LLM's Predicted Input: `f("123", "987")`

PPoT sample: `f("", "987")`

```
def f(x, y):
    tmp = ''.join(['0' if c == '9' else '9' for c in
          y[::-1]])
    if (x.isnumeric() and tmp.isnumeric()):
        return x + tmp
    else:
        return x
```

Example 36

Expected Output: `'ha'`

LLM's Predicted Input: `f('ha', 'a')`

PPoT sample: `f('ha', '')`

```
def f(text, chars):
    return text.rstrip(chars) if text else text
```

Example 66

Expected Output: `''`

LLM's Predicted Input: `f("hello", "world")`

PPoT sample: `f("", "world")`

```
def f(list):
    original = list[:]
    while len(list) > 1:
        list.pop(len(list) - 1)
        for i in range(len(list)):
            list.pop(i)
    list = original[:]
    if list:
        list.pop(0)
    return list
```

Example 172

Expected Output: `[]`

LLM's Predicted Input: `f([-1, -2, -3])`

PPoT sample: `f([-3])`

```
def f(array):
    for i in range(len(array)):
        if array[i] < 0:
            array.pop(i)
    return array
```

Example 193

Expected Output: `'1:1'`

LLM's Predicted Input: `f('1:1:1')`

PPoT sample: `f('1:1')`

```
def f(string):
    count = string.count(':')
    return string.replace(':', '', count - 1)
```

G Verifiers

The **PPoT** (as well as pass@k and Best-of-n) pipeline(s) rely on a verifier that takes in a program and outputs a score of that program. Up to this point, we have abstracted away what this verifier looks like. Formally, a verifier is simply a function $f : \mathcal{L}^* \to \mathbb{R}$, where $\mathcal{L}^*$ is the language over strings, that takes in a program and outputs a score.

In some tasks, the verifier is obvious: whenever a ground-truth $\boldsymbol{x}^\star$ is available, a reasonable verifier can be any function where $\forall \boldsymbol{x} \in \mathcal{L}^*, f(\boldsymbol{x}^\star) \geq \boldsymbol{x}$, i.e. it is a global maximum. This is the case of **GSM8k** and **CRUXEval**. In both cases we have access to the ground-truth. In **GSM8k** we are allowed to look at the ground-truth to verify, in **CRUXEval** we have access to both the Python function that produced the output and the desired output itself, and so we can easily execute the Python function using the LLM or **PPoT** generated input and verify whether that matches. In both of these cases, the verifier is simply $f : \boldsymbol{x} \mapsto [\![\boldsymbol{x} = \boldsymbol{x}^\star]\!]$, i.e. 1 if $\boldsymbol{x} = \boldsymbol{x}^\star$, 0 otherwise.

However, not all verifiers need to be Boolean. In the case of **Plot2Code**, it is unreasonable to say that the verifier must output 1 only if the generated plot is identical to the ground-truth. Ideally, the verifier should somehow capture the fact that small shifts or perturbations to the axis, size or plot positions are not as meaningful as the main content of the plot, such as the curve, labels and data points. To attempt to capture this, Wu et al. (2024) use a text match score to quantify how close to the ground-truth the generated plots are. The domain of the verifier in this case is $[0, 1]$.

H Subset sampling in CRUXEval

We describe the sampling procedure for subset-sampling in **CRUXEval** in Algorithm 3 and Algorithm 4.

Algorithm 3 Subset Sampling

Input Prompt t, sequence length n, number of samples k, subset samples m

```
samples ← []
probs ← {}
for j ∈ {1, 2, ..., ⌈k/m⌉}                          ┐ Repeat ⌈k/m⌉ times
    for i ∈ {1, 2, ..., n}                          ┐
        x_i ← sample(P_M(. | x_<i, t))              │ Sample from program m times
        probs[i] ← P_M(. | x_<i, t)                 ┘
    Prog ← compile_subset((x_1, ..., x_n), probs)   ┐
    for k ∈ {1, 2, ..., m}                          │
        obj ← subset_sample(Prog)                   │
        samples ← samples + [obj]                   ┘
return samples
```

Algorithm 4 `subset_sample`: Suffix-Masked Sequential Resampling

Input Token sequence $(x_1, \ldots, x_n)$, next-token distributions (`probs[1]`, ..., `probs[n]`)
Output Resampled sequence $(x'_1, \ldots, x'_l)$ where $l' \leq n$

$i \leftarrow 1$
while $i < n$
  $S_i \leftarrow \{x_j : j \geq i\}$ ▷ token types in original suffix
  $Q_i \leftarrow$ `probs[`i`]` restricted to S_i and renormalized
  $x'_i \leftarrow$ `sample`(Q_i)
  if $\exists\, j > i$ s.t. $x_j = x'_i$ in current sequence **then**
    delete $x_{i+1}, \ldots, x_{j-1}$ ▷ skip to matched position
  else
    replace $x_i \leftarrow x'_i$ ▷ no match; substitute in place
  $i \leftarrow i + 1$
return $(x'_1, \ldots, x'_l)$

I PPoT Scaling Laws

To quantify the value of PPoT samples relative to LLM samples, we fit a scaling law to the accuracy curves from Figure 3. For each configuration of k LLM samples boosted by m PPoT samples each (for a total of $k \cdot (1 + m)$ samples per problem), we model the error rate as a power law:

$$1 - \text{acc}(k; m) = a_m \cdot k^{-b_m} \tag{7}$$

where a_m and b_m are fit parameters that depend on the number of PPoT samples m. This functional form is standard in neural scaling laws (Kaplan et al., 2020). As argued in Schaeffer et al. (2025), the power law behavior in pass@k error can be attributed to a heavy-tailed distribution of task difficulty.

Figure 11 shows accuracy as a function of k for varying m, with scaling-law fits extrapolated beyond the data range. The corresponding log-log view in Figure 12 provides evidence that the curve fits the data well and that the power-law assumption is apt. In addition, Figure 12 reveals that PPoT samples increase the scaling exponent b_m (from $b_0 = 0.58$ for LLM-only to $b_{20} = 0.74$ for m=20), meaning the error decreases *faster* as we increase the number of PPoT samples.

Using the fitted LLM-only scaling law (m=0), we can compute the *LLM-equivalent* number of samples n^* needed to match the accuracy of any LLM+PPoT configuration:

$$n^* = \left(\frac{a_0}{1 - \text{acc}(k; m)} \right)^{1/b_0}. \tag{8}$$

The quantity $n^* - k$ then measures how many additional LLM samples would be needed to achieve the same accuracy without PPoT, providing a measure of the value of the $k \cdot m$ PPoT samples used.

We plot these quantities in Figure 6 to better understand the concrete benefit of PPoT. For example, using k=20 LLM samples and m=20 PPoT samples per LLM sample provides an accuracy boost equivalent to approximately 39 additional LLM samples, beyond the initial 20 LLM samples. Even a modest m=5 yields the equivalent of roughly 25 extra LLM samples. Because PPoT samples are computationally negligible compared to LLM samples, we can interpret the values in Figure 6 as the computational surplus achieved by PPoT.

J Related Work Full Discussion

We report on related work that has used probabilistic programming in the context of language models. To the best of our knowledge, our approach is the first one to propose interpreting the generated code by a language model as a probabilistic program for compute-efficient inference.

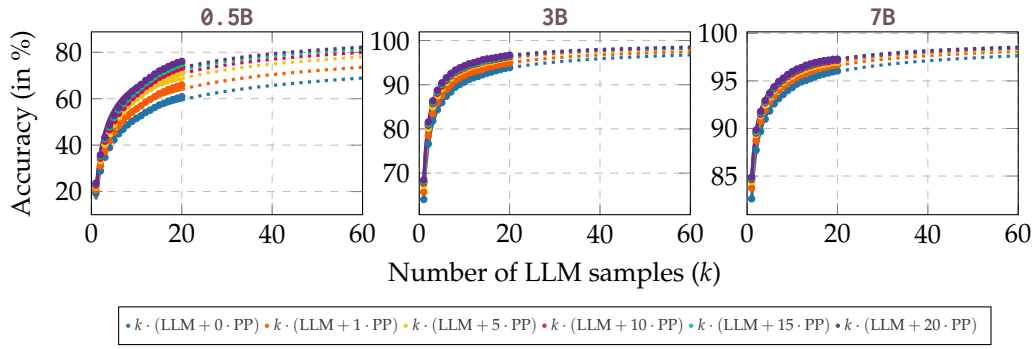


Figure 11: **Accuracy vs. number of LLM samples.** Points are observed data; curves are scaling-law fits $\text{acc} = 1 - a\,k^{-b}$. Solid lines span the data range; dotted lines are extrapolations. Each curve corresponds to k LLM samples boosted by $m \in \{0, 1, 5, 10, 15, 20\}$ PPoT samples each.

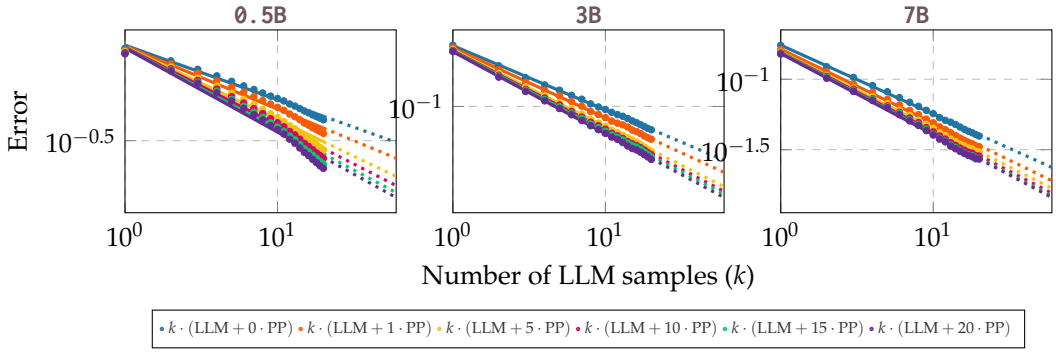


Figure 12: **Scaling-law fits in log-log space.** The error (1 − accuracy) follows a power law $a\,k^{-b}$. The steeper slopes for higher m show that PPoT samples improve the scaling exponent, not just the intercept.

Generating programs in specialized PPLs Recent language models have been developed to use external tools to reduce hallucinations and output accurate results. Probabilistic programming languages lie under the umbrella of external tools and several works have used language models to generate programs in specialized PPLs. Probabilistic-language-of-thought (Wong et al., 2023) uses language models to generate programs in Church (Goodman et al., 2008), Large Language Bayes (Domke, 2025) generates programs in Stan (Carpenter et al., 2017), NAVER (Cai et al., 2025) generates programs in Problog (Fierens et al., 2015) and RefineStat (Kanda et al., 2026) generates programs in PyMC (Abril-Pla et al., 2023) and proposes techniques to constrain the LLM output to well-formed probabilistic models (Karwowski et al., 2026). Since specialized PPLs have limited publicly available code, language models struggle to generate correct code in these languages. In our work, we get around this by directly interpreting the LLM generated Python code as a probabilistic program.

Probabilistic programming for constrained decoding and alignment Constrained decoding and alignment of language models have previously used the probabilistic programming perspective of separating modeling from inference (Korbak et al., 2022). Based on this, existing works have used different sampling approaches such as Sequential Monte Carlo (Loula et al., 2025; Lew et al., 2023) and Markov Chain Monte Carlo (Faria & Smith, 2025) to achieve control over outputs. These methods use probabilistic inference algorithms to guide sampling from the LLM, whereas our approach interprets the generated programs themselves as probabilistic programs.

Probabilistic techniques for LLM reasoning Several existing works have attempted to incorporate probabilistic facts into the workflow of language models. Dohan et al. (2022) uses probabilistic graphical models to characterize repeated interactions with a language model. Recent works have used LLM outputs together with their probabilities to improve reasoning for visual question answering using custom DSLs for modeling (Gupta & Kembhavi, 2023; Wan et al., 2025; Kamali & Kordjamshidi, 2025). Other work involves equipping reasoning chains with their associated probabilities to estimate the correctness of the LLM reasoning process (Feng et al., 2025; You et al., 2025; Cheng et al., 2026).

General programming techniques for LLMs and ML Finally, we note that our approach takes inspiration from other attempts to use programming techniques for formalize LLM algorithms, such as the prompting-as-programming approach of Beurer-Kellner et al. (2023) and the DSPy model (Singhvi et al., 2023; Khattab et al., 2023). Our work differs from this work in terms of being based on probabilistic programming. We also take inspiration from the large literature on declarative probabilistic programming techniques for LLMs (Richardson et al., 2025) and general ML (Xu et al., 2018; De Raedt & Kimmig, 2015; Manhaeve et al., 2018), which share a common algorithmic foundation to the kinds of probabilistic imperative programs used in our work (Holtzen et al., 2020).