

**LLM-based Efficient Exploration for Probabilistic Program Synthesis with
RefineStat**

by

Madhav Kanda

Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Advisor:

Associate Professor Sasa Misailovic

Abstract

Probabilistic programming offers a powerful framework for modeling uncertainty, yet statistical model discovery in this domain entails navigating an immense search space under strict domain-specific constraints. When small language models are tasked with generating probabilistic programs, they frequently produce outputs that suffer from both syntactic, and semantic errors, such as flawed inference constructs. Motivated by probabilistic programmers’ domain expertise and debugging strategies, we introduce `REFINESTAT`, a language model–driven framework that enforces semantic constraints ensuring synthesized programs contain valid distributions, well-formed parameters, and then applies diagnostic-aware refinement by resampling prior or likelihood components whenever reliability checks fail. We evaluate `REFINESTAT` on multiple probabilistic-programming code-generation tasks using smaller language models (SLMs) and find that it produces programs that are both syntactically sound and statistically reliable, often matching or surpassing those from closed-source large language models (e.g., OpenAI o3). Our code is available at <https://github.com/structuredllm/RefineStat>.

To my family, for their love and support.

Acknowledgements

I would like to thank my parents, my brother, and my sister-in-law for their constant presence through both the good and challenging phases of my journey. Despite being miles apart, my family always made me feel just a phone call away.

This thesis would not have been possible without the unwavering support and guidance of my advisor, Professor Sasa Misailovic, and Shubham Ugare. I am especially grateful to Sasa for his continuous encouragement and motivation throughout this journey—from a NeurIPS borderline rejection to an ICLR Oral acceptance. From taking me on a stroll on my first day at UIUC, to encouraging me through the NeurIPS deadline, to motivating me for resubmission to ICLR, to preparing me for the talk in Paris, and even patiently listening to my travel-related frustrations for ICLR—he has always been by my side, always encouraging me with unwavering positivity.

Beyond my family and mentors, I am deeply thankful to the people who made UIUC feel like a second home. Aryan and Khushi, for their support since the first semester; Dwip, for the many memorable hangouts and for being there for me in Seattle; Adharsh, Aval, Nirav, and Shaurya, for making the lab such a fun place to be. I am also grateful to the IITGN community at UIUC for the gatherings and for helping me settle in. I would like to extend my thanks to my ARC labmates for their valuable insights, especially Shubham, for guiding me both on and beyond the project.

I would also like to thank my Microsoft Research mentors—Sharad Agarwal, Rodrigo Fonseca, Alok Kumbhare, and Pedro Las-Casas for their guidance during my internship. I am also thankful to my Microsoft Research internship friends, with whom I shared some of my best moments.

Finally, I would like to thank my IITGN friends in the US — Anuj, Haikoo, Kush, Ksheer, Yeeshu, and Dhyey for always being by my side, for our regular weekend calls, and for their support during some of my toughest times.

Contents

Chapter 1	Introduction	6
Chapter 2	Background	9
2.1	Probabilistic Programming Diagnostics	11
Chapter 3	REFINESTAT	15
3.1	Semantically-Constrained Probabilistic Program Generation	17
3.2	Program Validation and Guided Resampling	20
Chapter 4	Illustrative Example	22
Chapter 5	Experimental Methodology	24
Chapter 6	Prompt Templates	26
6.1	Eight Schools	26
6.2	Dugongs	27
6.3	Surgical	28
6.4	GP	28
6.5	Peregrine	29
Chapter 7	Experimental Results	31
7.1	Improved Run Rate over Unconstrained and Syntax-driven Generation	31
7.2	Comparison of Generated Program Quality to Unconstrained Baseline	32
7.3	Comparison of Generated Program Quality to BoxLM	36
7.4	One-shot Generation with Frontier Models	37
7.5	Ablation Study	38
7.6	Token Efficiency	41
7.7	Generalizability Across Probabilistic Programming Backends	42
Chapter 8	Related Work	44
Chapter 9	Conclusions	46
	References	48

Chapter 1 Introduction

Scientific discovery often requires expressing complex systems as statistical models. Finding appropriate models that are both interpretable and computationally efficient is challenging. The vision of automating model discovery has a long-standing history. Past approaches have demonstrated success across various domains, such as identifying physical laws (Bongard and Lipson, 2007; McKinney et al., 2006; Linka, Pierre, and Kuhl, 2023), recovering the structure of nonlinear dynamical systems (Schmidt and Lipson, 2009), performing structure-aware nonparametric regression (Duvenaud et al., 2013), and tackling unsupervised learning problems (Roger Baker Grosse, 2014). However, they typically relied on significant manual effort – experts were required to define a domain-specific language (DSL) for representing models and engineer custom search algorithms for exploring compositions within that DSL.

Large Language Models (LLMs) have the potential to automate the model discovery by leveraging their extensive knowledge across various domains, enabling them to propose modeling approaches that were traditionally developed by human experts. However, using LLMs comes with significant challenges. Directly querying LLMs to generate statistical models often produces semantically flawed and unreliable programs, particularly in probabilistic programming languages like PyMC and NumPyro that evolve rapidly. These bugs hinder the correct execution of programs and constrain the effective exploration of the search space of working solutions. Further, running LLMs is costly, as expressive models (e.g., GPT-4) incur high API fees.

These limitations motivate Small Language Models (SLMs) as a practical alternative. Despite recent gains on coding tasks, they still produce *semantic* bugs code that runs but violates the intended statistical meaning. For example, in PyMC (Salvatier, Wiecki, and Fonnesbeck, 2016), an SLM can generate code that places variance where a standard deviation is expected – `pm.Normal(..., sigma=sigma**2)` instead of `pm.Normal(..., sigma=sigma)`. This change encodes the wrong statistical model, often inflating

uncertainty and even triggering errors such as `SamplingError`. In addition, SLMs may produce other semantic mistakes, such as using an invalid argument name `sd` in place of `sigma`, which raises a `TypeError` at model construction time. These cases illustrate the need for our constraints and Bayesian-workflow checks (Gelman, Vehtari, et al., 2020) to ensure correctness.

Our Work: REFINESTAT We present REFINESTAT, a novel probabilistic programming synthesis framework that efficiently guides a language model to generate probabilistic programs. REFINESTAT is the first to demonstrate that open-weight SLMs can synthesize *reliable* probabilistic programs in the Bayesian-workflow sense i.e., they satisfy standard checks, such as adequate effective sample size, low number of MCMC divergences and strong out-of-sample fit (Section Chapter 2 presents the full list).

REFINESTAT produces reliable statistical programs through a two-phase approach: (1) semantically constrained generation and (2) diagnostic-aware refinement (Section Chapter 3). In this context, semantically constrained denotes adherence to programming-language semantics (e.g., distribution validity, parameter consistency, proper data types), rather than the linguistic semantics of natural languages. Our semantic constraining ensures that synthesized probabilistic programs contain valid distributions with well-formed parameters, proper variable dependencies, and adherence to PyMC semantics. The diagnostic-aware refinement systematically resamples prior specifications or likelihood models when generated programs fail to meet established reliability criteria within the Bayesian workflow, thereby ensuring efficient search of probabilistic models using small language models.

We evaluate REFINESTAT on a suite of five representative probabilistic datasets, and five open-weight LLMs, with up to 8 billion weights. Our comparison shows that REFINESTAT significantly improves over directly querying LLMs in an unconstrained manner or only syntactic constraining with Syncode (Ugare, Suresh, et al., 2024). We show that the programs generated by REFINESTAT often pass the diagnostic metrics that

indicate high quality to represent and explain the data. We also show that the REFINESTAT’s performance is comparable to a recent LLM-based generation algorithm BoxLM (M. Y. Li, Fox, and Goodman, 2024), which uses two GPT-4 LLM instances to iteratively propose a likely program and refine it, respectively; yet REFINESTAT obtains those results with a single small language model.

Contributions: The main contributions of this paper are:

- **Approach:** We present REFINESTAT, a novel SLM-based framework for synthesis of probabilistic programs that are semantically correct and have high predictive performance.
- **Constrained decoding:** We propose using semantic constrained decoding to help generate syntactically and semantically valid probabilistic programs, at a small overall cost.
- **Iterative program search:** We present an iterative refinement loop that leverages a single, unmodified open-weights SLM to generate probabilistic programs with improved diagnostic metrics, refining statistical reliability by selectively resampling the likelihood and prior.
- **Evaluation:** We demonstrate that REFINESTAT performs significantly better than baseline language models, in terms of different diagnostic metrics, and in some cases performs equally well as GPT-4 and hand-written developer programs.

Chapter 2 Background

Language Models. Current autoregressive language models (LMs) operate on a vocabulary $V \subseteq \Sigma^*$ of tokens. A tokenizer converts an input prompt $O_0 \in \Sigma^*$ into a sequence of tokens t_1, t_2, \dots, t_k . The LM $M : V^* \rightarrow \mathbb{R}^{|V|}$ takes this sequence and outputs scores \mathcal{S} over the vocabulary: $\mathcal{S} = M(t_1, t_2, \dots, t_k)$. A softmax function transforms these scores into a probability distribution, from which t_{k+1} is sampled. More details on decoding and grammar-guided generation are provided below.

Decoding and Constraints. Various approaches for token selection include greedy decoding, sampling, and beam search, repeated until an end-of-sequence (EOS) token or another stopping criterion is met. In constrained decoding, we may need to exclude specific tokens at certain positions. This is achieved using a mask $m \in \{0, 1\}^{|V|}$, where 1 indicates a viable token and 0 a discarded one. Decoding methods can then be applied to $m \odot \text{softmax}(\mathcal{S})$.

Grammar-guided Generation Grammar-guided generation constrains model outputs to a formal grammar by using production rules of the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a sequence of nonterminals and *terminals* (the actual tokens or characters that appear in the final output). Most programming languages can be described using context-free grammar, with rules that apply to nonterminal symbols independent of their context. Grammar-guided generation ensures that LM outputs follow the syntactic structure required for, e.g., code generation or structured data formatting.

Bayesian Workflow. A robust Bayesian analysis follows an iterative workflow of model specification, posterior inference, diagnostic checking, and model comparison (Gelman, Vehtari, et al., 2020). This process can be summarized as: (1) specify the model (likelihood and priors, in our case using an LLM), (2) perform posterior inference, (3) conduct posterior predictive checks and convergence diagnostics, (4) if diagnostics pass, estimate out-of-sample fit (i.e., how well the model would predict data not used in fitting), and (5) compare and rank models by their relative out-of-sample performance (with uncertainty).

Further details about diagnostics and predictive evaluation are provided below.

At its core, a generative probabilistic program often follows a $\mathcal{D}\|\mathcal{P}\|\mathcal{L}$ structure: first fixing the observed data (\mathcal{D}), then sampling latent variables under the prior ($\mathcal{P} : p(z)$), and finally conditioning on the data via the likelihood ($\mathcal{L} : p(x | z)$).

Modern probabilistic programming languages such as Stan (Carpenter, Gelman, Matthew D Hoffman, et al., 2017a) and PyMC (Salvatier, Wiecki, and Fonnesbeck, 2016) streamline this cycle by automating MCMC sampling and providing integrated diagnostics. These include prior predictive checks, convergence measures (e.g., split- \hat{R} , effective sample size, BFMI, divergent NUTS transitions) (Vehtari, Gelman, Simpson, et al., 2021; Matthew D Hoffman, Gelman, et al., 2014; Betancourt, 2017), and predictive accuracy metrics such as PSIS-LOO (Vehtari, Gelman, and Gabry, 2017).

These diagnostics guard against model mis-specification and poor sampling behavior, ensuring that only well-calibrated models are considered for inference or downstream decision-making.

Probabilistic Programming. Statistical modeling aims to describe relationships between variables in data through joint probability distributions that capture both observed phenomena and underlying latent structure. In probabilistic modeling, we formalize this as a joint distribution $p(x, z|\eta)$, where $x = x_{1:n}$ represents n observed data points, $z = z_{1:m}$ denotes m latent variables, and η corresponds to fixed model parameters. The inferential goal is to compute the posterior distribution $p(z|x)$, which quantifies uncertainty in the latent variables conditional on observed data. Probabilistic programming languages (PPL) provide a flexible computational substrate for specifying joint distributions $p(x, z | \eta)$ as programs while leveraging automated inference methods (e.g., MCMC, variational inference) to compute the posterior $p(z | x)$ (Meent et al., 2021). Further details are provided below.

Probabilistic Programming Languages can be categorized as internal Domain-Specific Languages (DSLs), which embed within a host language and reuse its syntax and tooling

(e.g., PyMC (Salvatier, Wiecki, and Fonnesbeck, 2016), NumPyro (Phan, Pradhan, and Jankowiak, 2019), Pyro (Bingham et al., 2019)), or as external DSLs that define their own syntax and compiler (e.g., Stan (Carpenter, Gelman, Matthew D Hoffman, et al., 2017a)). This representation enables automated model specification while leveraging existing inference algorithms (Gordon et al., 2014).

2.1 Probabilistic Programming Diagnostics

Before presenting individual diagnostics, we briefly define a few recurring terms that are used throughout:

Chain: An independent run of the sampler that generates a sequence of draws from the posterior distribution. Multiple chains are typically run to verify that results do not depend on initialization.

Convergence: The state in which all chains are sampling from the same region of the posterior distribution. Lack of convergence suggests that the sampler has not fully explored the posterior.

Divergence: A warning issued by the Hamiltonian Monte Carlo (HMC) algorithm indicating that numerical integration failed to follow the posterior geometry accurately. Divergences often signal problematic parameterizations or highly curved posterior regions.

In our framework for valid statistical model synthesis, we employ several diagnostic metrics from probabilistic programming to ensure model validity. Below, we define each of these metrics formally.

Definition 1 (\widehat{R} Statistic) *The split- \widehat{R} statistic for parameter ϕ , denoted $\widehat{R}\phi$, measures the convergence of Markov chains in MCMC sampling by comparing the between-chain variance to the within-chain variance. Formally:*

$$\widehat{R}\phi = \sqrt{\frac{V}{W}} \tag{1}$$

where V is the variance between chain means and W is the average variance within chains. Values close to 1.0 indicate convergence, while higher values suggest poor mixing of chains.

Definition 2 (Effective Sample Size) *The effective sample size (ESS) measures the equivalent number of independent samples obtained from autocorrelated MCMC draws. For parameter ϕ , we define:*

$$\text{ESS}_{\text{bulk},\phi} = \frac{MN}{\tau_{\text{bulk},\phi}} \quad (2)$$

$$\text{ESS}_{\text{tail},\phi} = \frac{MN}{\tau_{\text{tail},\phi}} \quad (3)$$

where M is the number of chains, N is the number of draws per chain, and τ represents the autocorrelation time for bulk or tail estimates respectively. Bulk ESS evaluates sampling efficiency across the central mass of the posterior, while tail ESS focuses on the distribution tails.

Definition 3 (Divergent Transitions) *Divergent transitions, denoted $\text{divergences}(M)$ for model M , count the number of leapfrog steps in Hamiltonian Monte Carlo where the numerical approximation of Hamiltonian dynamics breaks down due to extremely high curvature in the posterior geometry. These indicate potential pathological geometries in the posterior distribution that may lead to biased inference.*

Definition 4 (Bayesian Fraction of Missing Information) *The Bayesian Fraction of Missing Information, $\text{BFMI}(M)$ for model M , is defined as:*

$$\text{BFMI}(M) = \frac{\text{Var}(\Delta E)}{\text{Var}(E)} \quad (4)$$

where E represents the energy (negative log probability density) and ΔE is the change in energy between consecutive HMC iterations. Low BFMI values indicate poor exploration of the target distribution.

Definition 5 (Pareto Shape Parameter) *The Pareto shape parameter $\hat{k}_i(M)$ for observation i in model M quantifies the reliability of importance sampling estimates used in PSIS-LOO cross-validation:*

$$\hat{k}_i(M) = \text{shape parameter of Pareto distribution fitted to importance weights for observation } i \quad (5)$$

Values $\hat{k}_i < 0.5$ indicate reliable estimates, while $\hat{k}_i > 0.7$ suggest unstable estimates that may require more robust computational approaches.

Definition 6 (Expected Log Pointwise Predictive Density) *The Expected Log Pointwise Predictive Density under Leave-One-Out cross-validation, $\widehat{\text{elpd}}(M)$ for model M , measures the model's out-of-sample predictive accuracy:*

$$\widehat{\text{elpd}}(M) = \sum_{i=1}^n \log p(y_i | y_{-i}) \quad (6)$$

where $p(y_i | y_{-i})$ is the predictive density for observation i after fitting the model to all other observations. Higher values indicate better predictive performance.

1. \widehat{R}_ϕ : Split- \widehat{R} statistic for parameter ϕ , measuring MCMC chain convergence.
2. $\text{ESS}_{\text{bulk},\phi}$: The effective bulk sample size for the parameter ϕ , estimating the sampling efficiency across the central mass of the posterior.
3. $\text{ESS}_{\text{tail},\phi}$: Tail effective sample size for parameter ϕ , measuring sampling efficiency in tails.
4. $\text{Divergences}(M)$: Count of divergent NUTS (Matthew D Hoffman, Gelman, et al., 2014) transitions in model M .
5. $\text{BFMI}(M)$: Bayesian Fraction of Missing Information for model M , assessing energy transition efficiency in Hamiltonian Monte Carlo (HMC) (Neal et al., 2011) algorithm.
6. $\hat{k}_i(M)$: Pareto shape parameter for observation i in PSIS-LOO (Pareto-smoothed importance sampling leave-one-out cross-validation), quantifying reliability of

importance sampling estimates. PSIS-LOO approximates exact LOO predictive densities by smoothing raw importance weights with a generalized Pareto fit to stabilize high-variance weights (Definition 7 below).

7. $\widehat{\text{elpd}}$: Expected Log Pointwise Predictive Density under Leave-One-Out cross-validation, measuring model’s out-of-sample predictive accuracy.

To evaluate out-of-sample predictive accuracy, we rely on the expected log pointwise predictive density under leave-one-out cross-validation (ELPD-LOO). A direct computation of LOO requires refitting the model n times (once for each observation), which is often too costly in practice. To avoid this, we use Pareto-smoothed importance sampling leave-one-out cross-validation (PSIS-LOO) (Vehtari, Gelman, and Gabry, 2017), which provides a fast approximation to ELPD-LOO and also gives diagnostics on influential observations via the Pareto \hat{k} values:

Definition 7 (PSIS-LOO (Vehtari, Gelman, and Gabry, 2017)) *Let $\mathcal{D} = \{y_i\}_{i=1}^n$ be the observed data and M a model. Draw $\{\theta^{(s)}\}_{s=1}^S$ from $p(\theta \mid \mathcal{D})$, compute raw importance weights $w_i^{(s)} = 1/p(y_i \mid \theta^{(s)})$, and let $\tilde{w}_i^{(s)}$ denote the Pareto-smoothed weights, obtained by replacing the largest tail weights with a generalized Pareto fit. Then the PSIS-LOO estimate is $\widehat{\text{elpd}}_{\text{PSIS-LOO}} = \sum_{i=1}^n \log \left[\frac{1}{S} \sum_{s=1}^S \tilde{w}_i^{(s)} p(y_i \mid \theta^{(s)}) \right]$.*

Instead of refitting the model n times, PSIS-LOO relies on a single full-data fit and stabilized importance weights. The generalized Pareto fit yields shape parameters k_i , which assess the reliability of the approximation; following standard guidance, the estimate is considered unreliable if about 20% of the k_i exceed 0.7.

Chapter 3 REFINESTAT

Figure 1 presents REFINESTAT’s two main ideas. First, we prune the search space of possible probabilistic programs by enforcing semantic validity during generation, mapping validation rules to nodes in the partial parse tree and resampling problematic program fragments when constraints are violated. Second, we implement diagnostic-aware refinement, systematically resampling components of statistically unsound models to satisfy Bayesian Workflow guidelines. This integrated approach aims to improve semantic correctness, statistical reliability, and yield strong predictive performance.

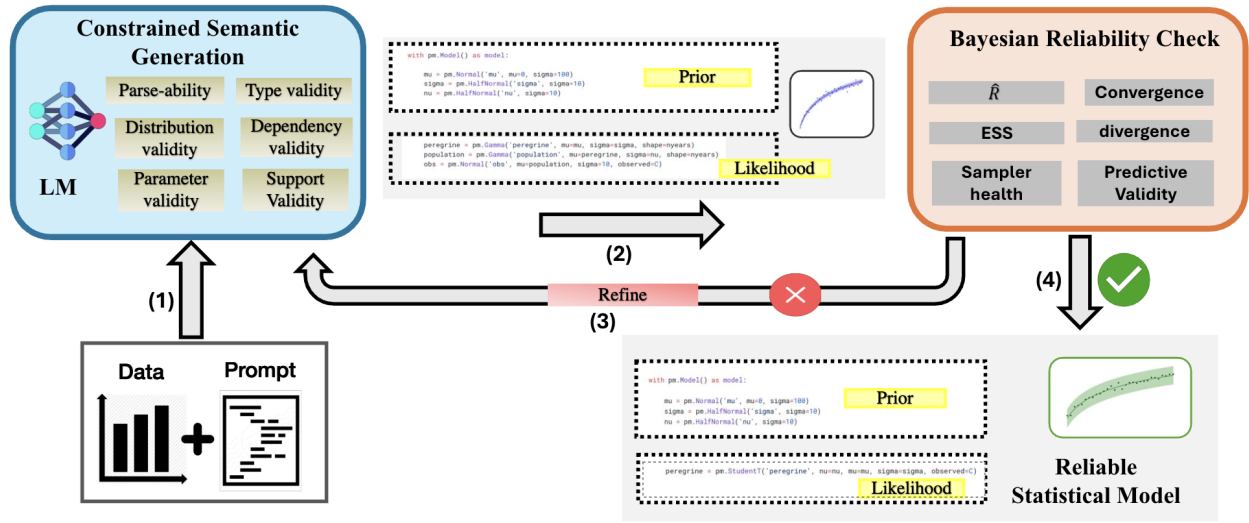


Figure 1: REFINESTAT workflow: (1) A user provides data and prompt to the language model, which generates a probabilistic program. (2) Constrained semantic decoding enforces syntactic and semantic validity of the generated program. (3) A Bayesian reliability check diagnoses convergence, divergences, and predictive validity. If failures are detected, the model is refined by backtracking and resampling priors or likelihoods. (4) Upon passing checks, we get final reliable probabilistic program.

Problem Statement. Let \mathcal{D} denote the dataset for a given statistical modeling task.

The objective of REFINESTAT is to construct a statistical model M in a probabilistic programming language that provides accurate predictive performance while quantifying uncertainty in a fully Bayesian manner.

Our approach to finding a model M that explains the data will follow the standard

Bayesian workflow (Gelman, Vehtari, et al., 2020) The key challenge to finding such a model M is to automatically compare various candidate models that an LLM produces. To automate this task, we will use a battery of diagnostics from statistical literature (Section Chapter 2), computed during the posterior inference in the standard Bayesian workflow (Gelman, Vehtari, et al., 2020).

Definition 8 (Bayesian Workflow Reliability Score) *Let \mathcal{M} be the set of candidate models, and fix thresholds $\alpha_R, \beta_{\text{bulk}}, \beta_{\text{tail}}, \gamma, L_{cd}, \epsilon$. For each $M \in \mathcal{M}$, we define seven indicator functions $s_j(M) \in \{0, 1\}$ by*

$$\mathbb{I}[A] = \begin{cases} 1, & \text{if event } A \text{ holds,} \\ 0, & \text{otherwise.} \end{cases}$$

and set

$$\begin{aligned} s_1(M) &= \mathbb{I}\left[\max_{\phi} \widehat{R}_{\phi}(M) \leq \alpha_R\right], & s_2(M) &= \mathbb{I}[\text{BFMI}(M) > \gamma], \\ s_3(M) &= \mathbb{I}\left[\min_{\phi} \text{ESS}_{\text{bulk}, \phi}(M) \geq \beta_{\text{bulk}}\right], & s_4(M) &= \mathbb{I}[\text{divergences}(M) = 0], \\ s_5(M) &= \mathbb{I}\left[\min_{\phi} \text{ESS}_{\text{tail}, \phi}(M) \geq \beta_{\text{tail}}\right], & s_6(M) &= \mathbb{I}\left[\frac{1}{n} \sum_{i=1}^n \mathbb{I}[\widehat{k}_i(M) \leq L_{cd}] \geq 1 - \epsilon\right], \\ s_7(M) &= \mathbb{I}[\widehat{\text{elpd}}(M) \text{ is finite}]. \end{aligned}$$

Then the reliability score is

$$\mathcal{B}(M) = \sum_{j=1}^7 s_j(M).$$

These diagnostics can be extracted directly from the MCMC engines (e.g. Stan (Carpenter, Gelman, Matthew D. Hoffman, et al., 2017b), PyMC (Salvatier, Wiecki, and Fonnesbeck, 2016)). Although $\widehat{\text{elpd}}$ provides a principled Bayesian measure of

out-of-sample predictive accuracy, its Monte Carlo estimate can be unreliable if the sampler has not fully converged or if the importance weights are unstable (Gelman, Carlin, et al., 1995). To mitigate these risks, we consider elpd estimates for models that satisfy standard convergence thresholds, thus ensuring that predictive comparisons rest on reliable posterior samples.

We require each model to pass *most* of these seven checks: if any check fails, the corresponding $s_j(M)$ is zero, and the total score reflects how many diagnostics remain satisfactory. The final check concerns the availability of the $\widehat{\text{elpd}}(M)$ estimate; if $\widehat{\text{elpd}}(M)$ cannot be computed or is infinite, then $s_{\text{ELPD}}(M) = 0$, and the model is treated as failing that diagnostic. A higher overall score indicates that more diagnostics have passed, so *when* $\widehat{\text{elpd}}(M)$ is available, the resulting estimate can be trusted with greater confidence. We consider a model reliable once its score exceeds a cutoff ζ . We use $\zeta = 5$ to allow marginal diagnostic failures while maintaining confidence in the reported $\widehat{\text{elpd}}$.

Definition 9 (Valid model space) *For a set of candidate models \mathcal{M} , a valid model space is:*

$$\mathcal{M}_{\text{valid}} = \{ M \in \mathcal{M} : \mathcal{B}(M) \geq \zeta \}.$$

Definition 10 (REFINESTAT Objective) *We finally define our objective to identify the model that attains the highest ELPD-LOO estimate within the space of models $\mathcal{M}_{\text{valid}}$:*

$$M^* = \arg \max_{M \in \mathcal{M}_{\text{valid}}} \widehat{\text{elpd}}(M),$$

3.1 Semantically-Constrained Probabilistic Program Generation

We formalize the generation of semantically valid probabilistic programs via iterative constrained sampling. Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S_0)$ be a context-free grammar with nonterminal symbols \mathcal{N} , terminal symbols \mathcal{T} , production rules \mathcal{P} , and start symbol S_0 .

For a partial program with parse tree κ , let $\mathcal{F}(\kappa)$ denote the set of program fragments, where each fragment $n \in \mathcal{F}(\kappa)$ is a rooted subtree of κ corresponding to a single syntactic

statement. *Validation functions* operate on a fragment n within program context $\pi \in \Pi$: $\Phi : \mathcal{F}(\kappa) \times \Pi \rightarrow \{0, 1\}$. These functions are conjunctions of individual correctness checks:

$$\Phi(n, \pi) = \phi^1(n, \pi) \wedge \phi^2(n, \pi) \wedge \dots \wedge \phi^m(n, \pi)$$

Each fragment thus corresponds to a single statement, possibly comprising multiple AST nodes.

Validity Predicates for Probabilistic Program Fragments. To ensure the correctness of synthesized probabilistic program fragments, we define three essential validation predicates. Let $\mathcal{F}(s)$ denote the set of all probability distribution functions invoked within fragment s . The validation predicates are:

1. **Parse-ability:** $\phi_1(s, \Pi) = 1$ if the fragment conforms to the grammar G .
2. **Distribution validity:** $\phi_2(s, \Pi) = \prod_{f \in \mathcal{F}(s)} \mathbf{1}\{f \in \mathcal{M}\}$ verifies that each probabilistic operation f exists in the available library \mathcal{M} of PPL.
3. **Parameter validity:** $\phi_3(s, \Pi) = \prod_{f \in \mathcal{F}(s)} \mathbf{1}\{P(f) \subseteq P_{\text{acc}}(f)\}$ confirms that operation parameters $P(f)$ adhere to the accepted specifications $P_{\text{acc}}(f)$, essential for maintaining probabilistic semantics. i.e. we ensure that the provided parameters for any distribution are correct according to the distribution’s specification. For instance, Figure 2 shows parameter ”sd” was invalid and resampled correctly as ”sigma”.
4. **Dependency validity:** $\phi_4(s, \Pi) = \prod_{v \in \text{Vars}(s)} \mathbf{1}\{\text{all dependencies of } v \text{ are defined before use}\}$ ensures that random variables are declared and initialized before they are referenced.
5. **Support validity:** $\phi_5(s, \Pi) = \prod_{f \in \mathcal{F}(s)} \mathbf{1}\{P(f) \in \text{Supp}(f)\}$ confirms that parameter values fall within the distribution’s support (e.g., variance > 0 , probabilities in $[0, 1]$).
6. **Type validity:** $\phi_6(s, \Pi) = \prod_{f \in \mathcal{F}(s)} \mathbf{1}\{\text{type}(P(f)) \in T(f)\}$ checks that each parameter $P(f)$ has the expected type from the specification $T(f)$, e.g., ensuring numeric values for scale parameters, or integer values for counts.

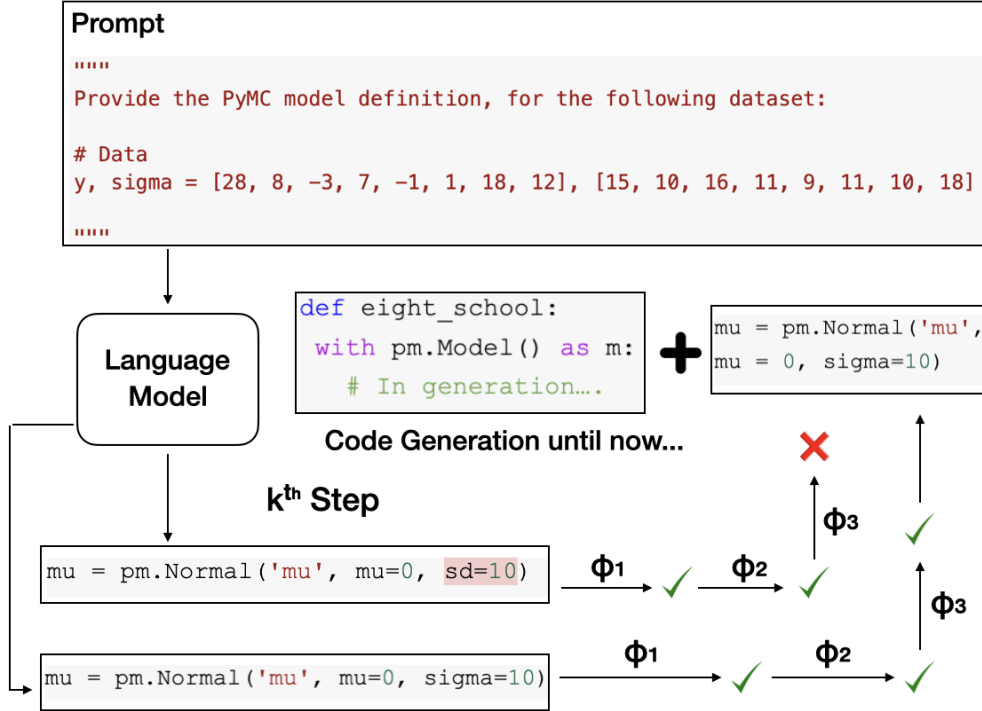


Figure 2: Constrained decoding in REFINESTAT fixing a `TypeError` from using `sd` instead of `sigma`, as illustrated in Section Chapter 1.

The final predicate $\Phi(s, \Pi) = \bigwedge_{i=1}^6 \phi_i(s, \Pi)$ ensures that generated fragments satisfy all requirements of the probabilistic programming language (1,4,6) and the Bayesian model (2,3,5). Our generation algorithm leverages these properties by maintaining a global symbol table $\Pi : \mathcal{A} \rightarrow \mathcal{M}$ mapping each alias $a \in \mathcal{A}$ to its module or namespace $m \in \mathcal{M}$.

Generation proceeds via local rejection sampling on S_N : we repeatedly sample $s \sim S_N$ until finding s^* where $\Phi(s^*, \Pi) = 1$. This iterative process continues until a termination fragment is generated, ensuring every component in the final probabilistic program satisfies all semantic constraints. Our local rejection sampling is token-efficient and we backtrack and precisely resample the tokens that correspond to the violation of the constraints. The approach is particularly effective for languages like PyMC, where maintaining consistent probabilistic variable scopes and dependencies is critical. Combining syntactic constraints with semantic validation enables efficient exploration of the program space while ensuring the probabilistic soundness of generated models.

3.2 Program Validation and Guided Resampling

Algorithm 1 REFINESTAT Synthesis via $\mathcal{D}\|\mathcal{P}\|\mathcal{L}$

Require: R_{\max} , α , β , $\{\tau_j\}_{j=1}^L$, K

- 1: $r \leftarrow 0$, $\ell \leftarrow 0$, $\mathcal{V} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $\mathcal{L} \leftarrow \emptyset$
- 2: **while** $r < R_{\max}$ **and** $|\mathcal{V}| < \beta$ **do**
- 3: Prog $\leftarrow \mathcal{D}\|\mathcal{P}\|\mathcal{L}$
- 4: **if** $\neg\Phi(\text{Prog})$ **then**
- 5: $r \leftarrow r + 1$; **continue**
- 6: compute diagnostics d_1, \dots, d_L on Prog
- 7: $P_{\text{pass}} \leftarrow |\{j : d_j \geq \tau_j\}|$
- 8: **if** $P_{\text{pass}} \geq K$ **then**
- 9: $\mathcal{V} \leftarrow \mathcal{V} \cup \{\text{Prog}\}$; **continue**
- 10: **if** $\ell < \alpha$ **then** \triangleright likelihood resampling
- 11: $\mathcal{L} \leftarrow L_{cd}(\mathcal{D}\|\mathcal{P})$
- 12: $\ell \leftarrow \ell + 1$
- 13: **else** \triangleright prior resampling
- 14: $\mathcal{P} \leftarrow L_{cd}(\mathcal{D})$
- 15: $r \leftarrow r + 1$
- 16: **return** $\arg \max \text{elpd}(M)$ over all $M \in \mathcal{V}$

Building on the validation predicate Φ , we now formalize our constrained generation and refinement methodology. We introduce the *constrained-decoding operator* $\mathcal{L}_{\text{CD}} : \mathcal{C} \rightarrow \mathcal{B}$, which implements our validation-guided sampling. Here, a *statement* is an individual syntactic unit, and a *code block* \mathcal{B} is a (possibly multi-statement) sequence of such statements. For any context \mathcal{C} , this operator returns a new code block \mathcal{B} satisfying $\Phi(\mathcal{C}\|\mathcal{B}) = 1$, where $\|$ denotes sequential concatenation of blocks; concretely, for two blocks

A and B , $A\|B$ is the program text formed by appending all statements of B immediately after those of A .

$\mathcal{D}\|\mathcal{P}\|\mathcal{L}$ denotes the full probabilistic program with data \mathcal{D} , prior \mathcal{P} , and likelihood \mathcal{L} . During refinement we perform *resampling* via two steps: $\mathcal{L} \leftarrow L_{cd}(\mathcal{D}\|\mathcal{P})$ (likelihood resampling), $\mathcal{P} \leftarrow L_{cd}(\mathcal{D})$ (prior resampling), guaranteeing replaced blocks remain semantically valid.

Before refinement begins, the data block \mathcal{D} is taken directly from the user prompt, while the initial prior and likelihood blocks are generated via constrained decoding, i.e., $\mathcal{P} \leftarrow \mathcal{L}_{CD}(\mathcal{D})$ and $\mathcal{L} \leftarrow \mathcal{L}_{CD}(\mathcal{D}\|\mathcal{P})$. Algorithm 1 synthesizes programs in two phases. First, it checks semantic correctness via Φ , ensuring parseability, distribution validity, and parameter consistency. Second, it evaluates Bayesian diagnostics d_1, \dots, d_L on the full program $\mathcal{D}\|\mathcal{P}\|\mathcal{L}$; at least K thresholds $\{\tau_j\}$ must be met to accept a candidate. As shown in Figure 1, if diagnostics fail, we perform one of two *resampling* steps to refine the program: (i) likelihood resampling replaces \mathcal{L} under the data–prior context, addressing convergence or sampler-health issues; (ii) prior resampling replaces \mathcal{P} under the data context, correcting prior-specification errors. We iterate until we collect β valid programs or exhaust the budget R_{\max} , and return the program maximizing $\widehat{\text{elpd}}$.

Chapter 4 Illustrative Example

We illustrate `REFINESTAT` on a standard Bayesian linear regression. Given a partial program P , we want to find a completion M that maximizes the Bayesian reliability score $B(M)$. This example shows how each semantic and diagnostic check prunes or refines candidates.

1. Partial Program P

At timestep t , `REFINESTAT` generates the following partial program:

```
with pm.Model() as linear_model:
    alpha = pm.Normal("alpha", 0, 10)
    beta = pm.Normal("beta", 0, 10)
    sigma = pm.HalfNormal("sigma", 5)
```

The LLM must complete the likelihood (y_{obs}).

2. LLM Proposals & Semantic Checks

Candidate likelihoods are checked against PPL semantics as shown in Table 1:

Table 1: Semantic filtering of LLM-proposed likelihoods.

Proposal	Check	Outcome
<pre>y_obs = pm.ExtNormal("y_obs", mu=alpha + beta * x, sigma=sigma, observed=y)</pre>	Distribution validity	Reject (hallucinated ExtNormal)
<pre>y_obs = pm.Normal("y_obs", mu=alpha + beta * x, sd=sigma, observed=y)</pre>	Parameter validity	Reject (deprecated sd vs. sigma)
<pre>y_obs = pm.Normal("y_obs", mu=alpha + beta * x, sigma=sigma, observed=y)</pre>	All checks	Accept

3. **Diagnostic Checks & Guided Resampling** We run NUTS on the accepted model and observe:

- $\widehat{R} = 1.2$ (too high),
- 100 divergences.

These failures reduce the Bayesian reliability score $B(M)$. REFINESTAT then resamples the likelihood or prior fragments (via the LLM) and retries inference until diagnostics (\widehat{R} , ESS, divergences, Pareto- k) fall within thresholds or the iteration limit is reached.

The final program M^* converges ($\widehat{R} \approx 1$), shows zero divergences, and yields reliable ELPD-LOO. By pruning invalid distributions early and resampling based on diagnostic triggers, REFINESTAT iteratively refines candidates into a high-scoring M^* with robust statistical reliability.

Chapter 5 Experimental Methodology

We use PyMC, a Python probabilistic programming library, to perform inference. We provide the same initial prompt while performing unconstrained generation and using REFINESTAT. We prompt the model by providing it with the dataset, the necessary library and the text query. The exact format of the prompt is provided in the next section.

As stated in Definition 8, we assess model reliability using standard Bayesian diagnostics (Vehtari, Gelman, Simpson, et al., 2021; Vehtari, Gelman, and Gabry, 2017; Gelman, Carlin, et al., 1995). Further details on hyperparameters and experimental setup are provided below.

We set the convergence threshold to $\alpha_R = 1.05$ for split- \widehat{R} , allow $\text{ESS}_{\text{bulk}} \geq \beta_{\text{bulk}} = 400$, and adopt a relaxed cutoff $\beta_{\text{tail}} = 100$ for ESS_{tail} to accommodate lower sampling efficiency in the tails. For leave-one-out validation, $\widehat{\text{elpd}}(M)$ must be finite, with at least $1 - \epsilon = 0.8$ of data points having Pareto shape values below $L_{\text{cd}} = 0.7$. These thresholds are used consistently when computing the reliability score across all models. We use STANDARD unconstrained generation as our baseline. Further, based on preliminary experiments we have chosen β to be 4, α to be 2, and R_{max} as 100 for all experimental purposes.

We run experiments on a 48-core Intel Xeon Silver 4214R CPU with 2 NVidia RTX A5000 GPUs. REFINESTAT is implemented using PyTorch (Paszke et al., 2019), and Itergen library (Ugare, Gumaste, et al., 2025) for refining the parser-guided LLM generation infrastructure. We run all experiments for 10 seeds to reduce result randomness, and use a temperature range of 0.2 to 0.4.

Datasets. We use five benchmark datasets from Stan PosteriorDB (Magnusson et al., 2024), mirroring the selection in prior research on automated statistical modeling (M. Y. Li, Fox, and Goodman, 2024):

- **Eight Schools (Rubin, 1981):** This dataset originates from a study commissioned by the Educational Testing Service, which examines the effects of coaching programs on

test performance.

- **Dugongs (Unit, n.d.[b]):** This dataset provides measurements on the ages and lengths of 27 dugongs.
- **Surgical (Unit, n.d.[a]):** This dataset comprises records on the number of cardiac surgeries performed on infants, along with the associated failure rates.
- **Peregrine (M Kery, 2011):** This dataset tracks the breeding trajectory of the peregrine falcon population in the French Jura region from 1964 to 2003.
- **GP:** This dataset contains simulated observations generated from a Poisson Gaussian Process.

Models. We experiment with a range of state-of-the-art LLMs, spanning multiple parameter scales including Qwen2.5 (code-specific) (B. Hui et al., 2024), models from the Llama series, DeepSeek, and Google’s CodeGemma. We have used a total of four models including, Llama3-8B (Grattafiori et al., 2024), CodeGemma-7B (Team et al., 2024), Qwen2.5-Coder-7B (B. Hui et al., 2024), and DeepSeek-R1-Distill-Qwen-7B (hereafter we refer to it as “DQ-7B”) (Guo et al., 2025).

Chapter 6 Prompt Templates

We use the same prompt across both the baseline, and REFINESTAT for experimentation purpose. To standardize the prompt across different datasets, we use a template in which the fields `{description}` and `{template_code}` are replaced with the dataset-specific description and code snippet, respectively.

```
Prompt Template

Template prompt:

# Complete the PyMC model definition within the 'with pm.Model() as m:'
  block below.
Your output must define a complete Bayesian model with appropriate priors,
  likelihood, and then sample the posterior using, `pm.sample(1000, tune =
  1000, chains = 4, return_inferencedata = True, idata_kwargs = {"
  log_likelihood": True})`. Do not include any extra commentary or text
  outside the code. Follow best practices for expert-level Bayesian
  modeling.

# Description: {Description}

{Template_Code}
```

Figure 3: Prompt template used across datasets in REFINESTAT.

Note: The placeholders `{description}` and `{template_code}` are automatically substituted for each dataset. Below are the `{description}` and `{template_code}` respectively for each dataset:

6.1 Eight Schools

Description: A hierarchical model for the 8-schools data.

Template Code

```
import pymc as pm
import numpy as np
import arviz as az
import matplotlib.pyplot as plt

# Data
y = np.array([28, 8, -3, 7, -1, 1, 18, 12])
sigma = np.array([15, 10, 16, 11, 9, 11, 10, 18])

with pm.Model() as m:
```

Figure 4: Template code for the Eight Schools dataset.

6.2 Dugongs

Description: A growth model for dugongs with missing data.

Template Code

```
import pymc as pm
import numpy as np
import arviz as az
import matplotlib.pyplot as plt

# Data
X = np.array([1, 1.5, 1.5, 1.5, 2.5, 4, 5, 5, 7, 8, 8.5, 9, 9.5, 9.5, 10,
              12, 12, 13, 13, 14.5, 15.5, 15.5, 16.5, 17, 22.5, 29, 31.5])
y = np.array([1.8, 1.85, 1.87, 1.77, 2.02, 2.27, 2.15, 2.26, 2.47, 2.19,
              2.26, 2.4, 2.39, 2.41, 2.5, 2.32, 2.32, 2.43, 2.47, 2.56, 2.65, 2.47,
              2.64, 2.56, 2.7, 2.72, 2.57])

with pm.Model() as m:
```

Figure 5: Template code for the Dugongs dataset.

6.3 Surgical

Description: The mortality rates in 12 hospitals performing cardiac surgery on babies.

Template Code

```
import pymc as pm
import numpy as np
import arviz as az
import matplotlib.pyplot as plt

# Given Data
N = 12 # Number of observations
n = np.array([47, 148, 119, 810, 211, 196, 148, 215, 207, 97, 256, 360])
r = np.array([0, 18, 8, 46, 8, 13, 9, 31, 14, 8, 29, 24])

with pm.Model() as m:
```

Figure 6: Template code for Surgical dataset.

6.4 GP

Description: Simulated data from a Poisson GP model.

Template Code

```
import pymc as pm
import numpy as np
import arviz as az
import matplotlib.pyplot as plt

# Given Data
N = 11 # Number of observations
x = np.array([-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10])
y = np.array([4.75906, 1.59423, 2.99548, 5.27501, 1.66472, 2.24347, 2.8914,
              4.08681, 4.60588, 0.802364, 3.92136])
k = np.array([40, 37, 29, 12, 4, 3, 9, 19, 77, 82, 33])

with pm.Model() as m:
```

Figure 7: Template code for the Peregrine dataset.

6.5 Peregrine

Description: Simulated population counts of peregrines in the French Jura over 9 years

Template Code

```
import pymc as pm
import numpy as np
import arviz as az
import matplotlib.pyplot as plt

# Data

nyears = 40 # Number of years
year = np.array([-0.95, -0.9, -0.85, -0.8, -0.75, -0.7, -0.65, -0.6, -0.55,
    -0.5, -0.45, -0.4, -0.35, -0.3, -0.25, -0.2, -0.15, -0.1, -0.05, 0, 0.05,
    0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7,
    0.75, 0.8, 0.85, 0.9, 0.95, 1])

C = np.array([27, 42, 35, 55, 61, 19, 41, 74, 43, 42, 73, 37, 48, 49, 19,
    72, 30, 18, 31, 71, 63, 51, 48, 73, 49, 54, 43, 59, 30, 24, 62, 55, 51,
    47, 14, 27, 45, 20, 26, 19])
N = np.array([43, 83, 53, 91, 95, 24, 62, 91, 64, 57, 97, 56, 74, 66, 28,
    92, 40, 23, 46, 96, 91, 75, 71, 100, 72, 77, 64, 68, 43, 32, 97, 92, 75,
    84, 22, 58, 81, 37, 45, 39])

with pm.Model() as m:
```

Figure 8: Template code for the Peregrine dataset.

Chapter 7 Experimental Results

7.1 Improved Run Rate over Unconstrained and Syntax-driven Generation

We conduct a comprehensive evaluation on diverse datasets, comparing `REFINESTAT` against both an unconstrained baseline and leading language models across a suite of diagnostic and performance metrics. We used identical prompts across our framework, the Standard baseline (unconstrained), syntactically-constrained tool `Syncode` (Ugare, Suresh, et al., 2024), and `REFINESTAT`.

Table 2 presents Run rates across different temperature settings. Run rate is the fraction of programs that successfully produce the samples from the posterior distribution. The problems that do not run successfully include those with runtime errors such as (1) numerical errors, e.g., `inf/nan`, (2) sampling issues due to unlikely prior parameterization, (3) other sampling warnings, e.g., failed to initialize chains, (4) static compilation issues.

Table 2: Run rates for Standard, `SYNCODE`, and `REFINESTAT` by temperature

Temp.	Standard	<code>SYNCODE</code>	<code>REFINESTAT</code>
0.2	0.10	0.21	0.45
0.3	0.11	0.21	0.50
0.4	0.11	0.21	0.50

`REFINESTAT` achieves success rates approximately 40 percentage points higher than the Standard baseline and 30 points higher than `Syncode`, demonstrating that our validation-guided approach substantially enhances code generation reliability by mitigating both syntactic and

semantic error sources. These results show that `REFINESTAT` significantly enhances code generation reliability.

We categorized the failures by their root causes in different methods found during the run rate experiment:

- The Standard baseline exhibited frequent syntax errors (e.g., unmatched delimiters, missing imports) and invalid API calls.
- Syncode eliminated many basic syntactic mistakes but still suffered semantic errors, such as incorrect distribution parameter names, type mismatches, referring to deprecated API functions (e.g., calling `pm.sample_prior` from an earlier PyMC release), and inventing non-existent methods like `pm.random_coefs`.
- RefineStat, in contrast, often produced models whose samplers failed to explore the correct posterior modes, leading to chains stuck in low-density regions or divergent transitions, failures stemming from the model definitions rather than our decoding framework; REFINESTAT reduced both syntactic and semantic errors and avoided sampler pathologies by enforcing grammar and parameter validity during decoding.

7.2 Comparison of Generated Program Quality to Unconstrained Baseline

To evaluate semantic correctness and diagnostic robustness, we compared programs generated by the base language model with those from REFINESTAT under identical prompts. Since we observed that REFINESTAT consumes almost twice the number of tokens used by the Baseline (see details below), we run baseline models five times with different seeds ($2.5\times$ tokens more than REFINESTAT) and compare the best program based on Bayesian Reliability Score and ELPD LOO to a single REFINESTAT run. We repeat this process five times to compute the mean and standard deviation for all metrics. In Table 3, we report five representative metrics drawn from these diagnostics (for space reasons). Bold entries denote cases where REFINESTAT outperforms the corresponding baseline. Since the Reliability Score aggregates multiple diagnostic metrics, it is highlighted most frequently; when Reliability Scores are similar, we instead emphasize differences in ELPD LOO. The symbol \times indicates that no valid program was produced—i.e., the method failed to explore the search space sufficiently to yield a correct result.

Table 3: Comparison of Diagnostic Scores and ELPD-LOO for Standard vs. REFINESSTAT

Dataset	Model	Variant	Reliab. Scr. \uparrow		\widehat{R} \downarrow		ESS Bulk \uparrow		Diverg. \downarrow		Pareto k \downarrow		ELPD LOO \uparrow		
			Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	
8 Schools	Meta-LLama-3-8B	Standard	7.00	0.00	1.00	0.00	2261.00	0.00	0.00	0.00	0.00	0.00	0.00	-31.70	0.00
		RefineStat	7.00	0.00	1.00	0.00	2303.00	768.76	0.00	0.00	0.00	0.05	-31.77	0.61	
	CodeGemma-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	7.00	0.49	1.00	0.00	2573.00	527.68	0.00	1.97	0.00	0.06	-31.46	1.84	
	Qwen-Coder-7B	Standard	3.80	1.30	1.02	0.01	223.25	80.43	92.75	31.38	0.22	0.21	-31.31	0.68	
		RefineStat	5.00	0.45	1.01	0.01	256.50	883.83	34.50	86.89	0.00	0.05	-30.80	0.05	
	DQ-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	5.00	0.98	1.02	0.02	219.00	2176.00	102.00	75.36	0.00	0.00	-30.68	0.11	
Dugongs	Meta-LLama-3-8B	Standard	7.00	0.00	1.00	0.00	2167.67	884.79	0.00	0.00	0.01	0.02	-7.66	30.39	
		RefineStat	7.00	0.00	1.00	0.00	1696.00	284.33	0.00	0.00	0.00	0.02	8.42	24.51	
	CodeGemma-7B	Standard	5.70	2.30	1.04	0.06	1527.33	1325.54	285.67	494.79	0.00	0.00	3.90	7.71	
		RefineStat	7.00	0.00	1.00	0.00	1908.00	2066.23	0.00	0.00	0.04	0.02	8.07	15.42	
	Qwen-Coder-7B	Standard	7.00	0.00	1.00	0.01	1788.67	123.43	0.00	0.00	0.04	0.00	8.15	0.29	
		RefineStat	7.00	0.00	1.00	0.00	1683.00	148.16	0.00	0.00	0.04	0.02	8.29	0.05	
	DQ-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	7.00	0.00	1.00	0.00	2376.00	149.61	0.00	0.00	0.00	0.03	8.35	0.06	
Peregrine	Meta-LLama-3-8B	Standard	6.00	0.00	1.00	0.00	3774.00	0.00	7.00	0.00	0.00	0.00	-184.96	0.00	
		RefineStat	7.00	0.00	1.00	0.00	3574.00	428.26	0.00	0.00	0.00	0.00	-173.00	4.91	
	CodeGemma-7B	Standard	7.00	0.00	1.00	0.00	4261.00	0.00	0.00	0.00	0.00	0.00	-172.91	0.00	
		RefineStat	6.50	0.53	1.00	0.00	2930.00	1343.79	0.50	0.53	0.00	0.00	-129.93	3.91	
	Qwen-Coder-7B	Standard	7.00	0.00	1.00	0.00	4238.00	0.00	0.00	0.00	0.00	0.00	-173.11	0.00	
		RefineStat	7.00	0.00	1.00	0.00	4679.00	88.73	0.00	0.00	0.00	0.00	-172.98	0.12	
	DQ-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	6.50	0.53	1.01	0.01	963.50	436.70	25.50	27.26	0.00	0.00	-114.29	2.76	
Surgical	Meta-LLama-3-8B	Standard	5.50	1.30	1.01	0.01	1159.75	1202.11	1066.50	1267.61	0.65	0.44	-31.59	20.02	
		RefineStat	6.00	0.77	1.00	0.00	579.00	1272.12	0.00	5.75	0.00	0.36	-46.73	36.63	
	CodeGemma-7B	Standard	5.50	0.70	1.00	0.00	1230.00	503.46	6.00	5.66	0.46	0.65	-42.02	5.75	
		RefineStat	6.00	0.49	1.00	0.00	2026.00	450.46	0.00	0.49	0.00	0.12	-45.55	381.60	
	Qwen-Coder-7B	Standard	6.30	1.50	1.01	0.02	1332.75	784.13	37.50	75.00	0.23	0.46	-44.48	4.35	
		RefineStat	7.00	0.41	1.00	0.00	1642.00	1350.46	0.00	0.00	0.00	0.38	-46.55	2.79	
	DQ-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	7.00	0.50	1.00	0.01	2800.00	1182.68	0.00	0.00	0.00	0.37	-46.51	3.49	
GP	Meta-LLama-3-8B	Standard	3.00	0.00	4.13	0.00	4.00	0.00	1634.00	0.00	0.00	0.00	-21.61	0.00	
		RefineStat	6.00	0.49	1.00	0.00	1710.00	668.57	13.00	12.39	0.09	0.08	-152.30	139.07	
	CodeGemma-7B	Standard	7.00	0.00	1.00	0.00	6034.00	0.00	0.00	0.00	0.18	0.00	-154.42	0.00	
		RefineStat	7.00	0.98	1.00	0.00	1752.00	1004.88	0.00	2.46	0.00	0.49	-22.76	126.08	
	Qwen-Coder-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
	DQ-7B	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
		RefineStat	6.50	0.53	1.01	0.00	816.50	89.27	30.50	32.61	0.00	0.00	-23.39	1.14	

Except for one instance, REFINESSTAT matches or exceeds the Standard Baseline in terms of the Bayesian Workflow Reliability score, and in some cases achieves up to twice the reliability. Moreover, REFINESSTAT delivers substantially better performance on

individual diagnostics, particularly divergences. Notably, DQ-7B, which failed on every dataset under the Standard Baseline, succeeded on all datasets when augmented with REFINESTAT. For example, on the Surgical dataset with Meta-Llama, the Standard Baseline produces over 1000 divergences, while REFINESTAT produces none.

We observed that in cases where the Standard baseline attains a higher ELPD-LOO than REFINESTAT, closer inspection of the diagnostics exposes unreliable sampling. For instance, on the Meta-Llama GP task the Standard model achieves a superior ELPD score, but exhibits split- $\hat{R} = 4.13 \gg 1$ and a low reliability score (3), indicating severe convergence issues. In contrast, REFINESTAT may report a marginally lower ELPD yet maintains $\hat{R} \approx 1$ and a higher reliability score, reflecting trustworthy posterior estimates. Furthermore, REFINESTAT delivers markedly lower variability in key diagnostics such as \hat{R} and the number of divergent transitions demonstrating its consistency and robustness. We further illustrate the types of structural changes introduced during refinement below.

In addition to refining a fixed model skeleton, REFINESTAT can introduce qualitatively new structural components during the resampling process. When the model families and diagnostic feedback indicates low reliability, REFINESTAT may alter the functional form of the model, add new parameters, or change the dependency structure to explore alternatives with higher Bayesian reliability scores.

Below, we present three programs generated for the same dataset and initialization. These examples illustrate how REFINESTAT’s refinement process can modify terms in the likelihood, expand the parameter space, or even change which variable is treated as observed.

Program 1: Standard linear regression

```
alpha = pm.Normal("alpha", mu=0, sigma=10)
beta  = pm.Normal("beta", mu=0, sigma=10)
sigma = pm.HalfNormal("sigma", sigma=1)

mu = alpha + beta * year
likelihood = pm.Normal("likelihood", mu=mu, sigma=sigma, observed=C)
```

Figure 9: Program 1: Standard linear regression.

Program 2: Nonlinear likelihood and an additional prior

```
alpha = pm.Normal("alpha", mu=0, sigma=10)
beta  = pm.Normal("beta", mu=0, sigma=10)
gamma = pm.Normal("gamma", mu=0, sigma=10)
sigma = pm.HalfNormal("sigma", sigma=10)

likelihood = (alpha + beta * year + gamma * year**2) * C + sigma * N
observed = pm.Normal("observed", mu=likelihood, sigma=sigma, observed=C)
```

Figure 10: Program 2: Nonlinear likelihood with an additional prior.

Program 3: Changing the response variable and dependency structure

```
alpha = pm.Normal("alpha", mu=0, sigma=10)
beta  = pm.Normal("beta", mu=0, sigma=10)
gamma = pm.Normal("gamma", mu=0, sigma=10)
sigma = pm.HalfNormal("sigma", sigma=10)

y_pred = alpha + beta * year + gamma * C
y_obs  = pm.Normal("y_obs", mu=y_pred, sigma=sigma, observed=N)
```

Figure 11: Program 3: Changing the response variable and dependency structure.

Summary of Structural Differences. Program 1 uses a standard linear regression and models \mathbf{C} as a linear function of year . Program 2 expands the model by introducing a new parameter (gamma) and applying a nonlinear transformation involving year^2 , \mathbf{C} , and \mathbf{N} , leading to a different generative process. Program 3 changes the response variable entirely, modeling \mathbf{N} instead of \mathbf{C} , and modifies the dependency graph by incorporating \mathbf{C} as a covariate in the linear predictor.

These examples show that REFINESTAT is capable of exploring alternative model families and introducing new structure, such as additional priors, nonlinear link functions, or altered observational targets whenever such modifications remain semantically valid.

7.3 Comparison of Generated Program Quality to BoxLM

Table 4: Comparison of ELPD LOO scores with BoxLM (M. Y. Li, Fox, and Goodman, 2024), REFINESTAT using DQ-7B, and Expert values

Dataset	Expert	REFINESTAT w/ DQ-7B(mean \pm std)	BoxLM w/ GPT-4	OpenAI-o3 (mean \pm std)
Eight schools	-30.70	-30.68 \pm 0.11	-30.42	-30.74 \pm 0.07
Dugongs	22.43	8.35 \pm 0.06	23.40	22.83 \pm 8.12
Peregrine	-112.60	-114.29 \pm 2.76	-173.11	-133.29 \pm 10.33
Surgical	-39.73	-46.51 \pm 3.49	-38.03	-38.73 \pm 0.51
GP	-26.53	-23.39 \pm 1.14	-	-34.95 \pm 13.28

We compare REFINESTAT’s performance using the DQ-7B model (averaged over five runs) against three baselines: the *Expert* stan programs from PosteriorDB (Magnusson et al., 2024), the BoxLM system introduced by M. Y. Li, Fox, and Goodman (2024), and programs generated by OpenAI o3. Since the code for BoxLM is not publicly available, we rely on the reported numbers from their paper for comparison. Because the dataset we use

for the GP task is not included in BoxLM’s evaluation suite, we omit their result for that dataset.

Table 4 presents the ELPD LOO scores, showing that our framework consistently outperforms both BoxLM and OpenAI o3 on the PEREGRINE dataset, and surpasses OpenAI o3 on the GP task, while matching the performance of expert-written programs in most cases. Across the remaining datasets, our approach performs comparably to other baselines, with the exception of DUGONGS, where performance is slightly lower. These results demonstrate that REFINESTAT achieves performance comparable to, and in several cases better than, large language models like OpenAI o3 and multi-agent frameworks such as BoxLM.

7.4 One-shot Generation with Frontier Models

We additionally evaluate the ability of a frontier model, Claude Opus 4.7 (release: April 16, 2026), to generate probabilistic programs in a single pass using the same prompt template, without any refinement or feedback loop. Our experiments were conducted in Fall 2025, whereas prior work such as BoxLM (Stanford, 2024) relied on multi-agent systems to achieve similar probabilistic program synthesis capabilities. In contrast, modern frontier models are now capable of generating comparable programs in a single pass, illustrating the rapid progress in large language model capabilities within a short span of time.

We observe that the generated programs are syntactically correct, compile successfully, and, in our experiments, yield stable and reliable inference across the evaluated benchmarks. In all cases, the resulting programs achieve ELPD-LOO scores close to those of expert-written models, demonstrating strong one-shot generation capability.

These results highlight the rapid progress of frontier models for probabilistic program synthesis, where capabilities have improved substantially within the span of a single year. At the same time, our approach is designed with a different objective: rather than relying on a single high-capacity model, REFINESTAT provides a structured, feedback-driven

refinement process that enables smaller models to achieve comparable performance. This distinction is particularly important in settings where model access, cost, or deployment constraints limit the use of such frontier models, or where iterative control and interpretability of the modeling process are desired. This perspective is aligned with recent trends emphasizing the effectiveness and growing relevance of smaller models (Belcak et al., 2025).

7.5 Ablation Study

Effectiveness of Semantic Validation Components. To evaluate the contribution of each validity predicate within REFINESTAT, we perform an ablation study by systematically disabling one component at a time and measuring the resulting compilation rate. This analysis is conducted across all models with ten different random seeds to ensure robustness. Notably, removing all validation predicate reduces the system to the behavior of SYNCODE.

Table 5: Ablation Study: Impact of Semantic Validation Checks on Run rate

#	Method	Run %	Δ Run
1	REFINESTAT (all components)	50.0%	-
2	w/o Parameter validity	35.5%	-14.5%
3	w/o Distribution validity	26.5%	-9%
4	w/o Parse-ability	21.0%	-5.5%
5	w/o grammar-guided generation	11.0%	-10.0%

Table 5 shows that parameter validity emerges as the most critical component, its removal results in a substantial drop of 14.5 percentage points in compilation success. All checks contribute meaningfully to overall performance: omitting distribution validity or parse-ability consistently reduce compilation rates (-9% and -5.5%, respectively),

underscoring the complementary role these validations play in ensuring soundness of generated programs.

Memorization Effect. A number of studies (Yihong Dong et al., 2024; Golchin and Surdeanu, 2025; Golchin and Surdeanu, 2024; Y. Li, 2023) have highlighted concerns about the memorization effect in large language models, where models may reproduce previously seen content rather than demonstrating genuine synthesis. (Kong, Xie, and S. Liu, 2025) addresses this issue by proposing code mutation to reveal potential memorization in program repair tasks. Inspired by this perspective, we introduce dataset and prompt modifications designed to preserve the semantics of the data while changing its presentation. We apply two systematic prompt modifications across all benchmarks when evaluating Meta-Llama 3-8B, with details provided below.

To further stress-test our approach against memorization, we performed two controlled prompt modifications across all datasets using Meta-Llama 3-8B.

Anonymized Prompt (REFINESTAT-AP): All metadata and dataset names were removed, leaving only the raw dataset.

Syntactic Obfuscation (REFINESTAT-SO): All numerical values were transformed into exponential notation (e.g., 3.28e2 instead of 328) to prevent exact string matches with any potential training data.

Anonymized Prompt and Syntactic Obfuscation (REFINESTAT-AP-SO): Combined variant using both Anonymized Prompt, and Syntactic Obfuscation.

Table 6: Comparison of Diagnostic Scores and ELPD-LOO for REFINESTAT variants.

Dataset	Variant	Reliab. Score \uparrow		$\widehat{R} \downarrow$		ESS Bulk \uparrow		Divergences \downarrow		Pareto $k \downarrow$		ELPD LOO \uparrow	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Dugongs	REFINESTAT	7.00	0.00	1.00	0.00	1696.00	284.33	0.00	0.00	0.00	0.02	8.42	24.51
	REFINESTAT-AP	7.00	0.00	1.00	0.00	1073.00	317.84	0.00	0.00	0.04	0.00	-0.17	4.24
	REFINESTAT-SO	7.00	0.00	1.00	0.00	1753.50	68.95	0.00	0.00	0.04	0.00	8.31	0.09
	REFINESTAT-AP-SO	7.00	0.00	1.00	0.00	2257.00	731.23	0.00	0.00	0.00	0.00	1.79	7.09
Eight Schools	REFINESTAT	7.00	0.00	1.00	0.00	2303.00	768.76	0.00	0.00	0.00	0.05	-31.77	0.61
	REFINESTAT-AP	7.00	0.00	1.00	0.00	2926.00	541.38	0.00	0.00	0.00	0.00	-31.62	0.79
	REFINESTAT-SO	7.00	0.00	1.00	0.00	2470.50	32.61	0.00	0.00	0.06	0.06	-31.61	0.01
	REFINESTAT-AP-SO	7.00	0.00	1.00	0.00	2187.50	252.83	0.00	0.00	0.00	0.00	-31.60	0.04
Peregrine	REFINESTAT	7.00	0.00	1.00	0.00	3574.00	428.26	0.00	0.00	0.00	0.00	-173.00	4.91
	REFINESTAT-AP	7.00	0.00	1.00	0.00	4057.00	811.85	0.00	0.00	0.00	0.00	-173.14	65.91
	REFINESTAT-SO	7.00	0.00	1.00	0.00	1812.50	24.05	0.00	0.00	0.00	0.00	-132.88	8.20
	REFINESTAT-AP-SO	6.00	0.00	1.00	0.00	1812.50	24.05	0.00	0.00	0.00	0.00	-140.88	8.20
GP	REFINESTAT	6.00	0.49	1.00	0.00	1710.00	668.57	13.00	12.39	0.09	0.08	-152.30	139.07
	REFINESTAT-AP	7.00	0.00	1.00	0.00	2283.00	519.55	0.00	0.00	0.00	0.04	-21.24	2.58
	REFINESTAT-SO	7.00	0.00	1.00	0.00	1135.00	0.00	0.00	0.00	0.00	0.00	-24.99	0.00
	REFINESTAT-AP-SO	6.50	0.53	1.00	0.00	1140.00	251.23	69.00	73.76	0.00	0.00	-23.01	0.47
Surgical	REFINESTAT	6.00	0.77	1.00	0.00	579.00	1272.12	0.00	5.75	0.00	0.36	-46.73	36.63
	REFINESTAT-AP	7.00	0.49	1.01	0.00	640.00	635.48	0.00	0.00	0.00	0.25	-46.71	96.47
	REFINESTAT-SO	7.00	0.00	1.01	0.01	1311.00	742.99	0.00	0.00	0.04	0.04	-65.22	19.85
	REFINESTAT-AP-SO	7.00	0.00	1.01	0.01	1139.00	638.22	0.00	0.00	0.04	0.04	-69.27	24.20

As shown in Table 6, both variants match the original REFINESTAT on reliability, convergence, and predictive metrics. The table also reports a combined variant using both modifications, which performs comparably. This consistency suggests that REFINESTAT’s effectiveness stems from learning from the provided data rather than memorization.

Additionally, PosteriorDB provides limited ground-truth PyMC programs: only the Eight Schools model is available, and it targets an outdated version of PyMC, while the remaining programs are provided in Stan. While these studies are limited in scope, and measuring the memorization effect is still an open problem, they give an indication that REFINESTAT’s effectiveness may not be the consequence of just memorization.

7.6 Token Efficiency

To evaluate the computational cost associated with our framework, we measure the number of tokens consumed in generating a program under Itergen, Baseline (Unconstrained generation), REFINESTAT without Refinement Loop (REFINESTAT w/o RL), and REFINESTAT using Meta-LLama-3-8B. The token usage across five runs is recorded for each of these methods, from which we report the mean and standard deviation.

The Table 7 presents the results across models and datasets, with the final column (“Token Ratio”) reporting the ratio of token usage by REFINESTAT relative to the baseline. On average, REFINESTAT consumes twice the number of tokens consumed by Baseline. While this reflects the added cost of our refinement mechanism, the overhead varies across settings. For instance, in some cases such as Dugongs, REFINESTAT uses fewer tokens than the baseline due to early convergence. Conversely, high multipliers (e.g., Eight Schools, GP) reflect continued refinement due to unmet stopping conditions, even if the generated program is already of high quality.

Table 7: Comparison of Itergen, Baseline, and REFINESTAT Variants with Multipliers.

Dataset	Itergen		Baseline		REFINESTAT w/o RL		REFINESTAT		Token
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Ratio
Eight Schools	98.8	5.3	606.8	386.3	147.6	36.7	1503.0	409.5	2.5x
Dugongs	209.3	225.9	747.4	495.0	294.4	86.3	419.0	87.1	0.6x
GP	131.3	8.9	663.6	410.9	155.0	21.2	1660.0	253.5	2.5x
Peregrine	132.5	21.4	884.0	538.9	180.8	103.6	1740.0	418.0	2.0x
Surgical	97.3	4.3	624.4	260.6	113.2	9.2	1275.0	1348.1	2.0x
Average Token Ratio									1.9x

7.7 Generalizability Across Probabilistic Programming Backends

Although our primary evaluation uses PyMC, the design of REFINESTAT is not specific to any single probabilistic programming library. To show generalizability of the framework, we apply REFINESTAT to NumPyro using the same prompting setup and the Qwen-2.5-Coder-7B model.

Across temperatures, REFINESTAT substantially improves the fraction of programs that successfully compile and execute. Table 8 shows that the run rate more than doubles relative to Standard unconstrained decoding, consistent with the improvements in PyMC3 experiment (Table 2).

Table 8: Run-rate comparison when using NumPyro as the inference backend.

Temp	Standard	REFINESTAT
0.2	0.17	0.34
0.3	0.15	0.35
0.4	0.15	0.33

Table 9 reports the metrics for NumPyro generated programs across all benchmarks, as those in Table 3 (for PyMC3). For every dataset, REFINESTAT attains equal or higher reliability than the Standard baseline. When both approaches achieve similar reliability, REFINESTAT yields comparable or improved predictive performance. Notably, in the GP task, Standard decoding fails to produce any valid model, whereas REFINESTAT consistently generates executable programs with stable diagnostics. Overall, these results show that REFINESTAT’s semantic filtering and diagnostic-aware refinement generalize across PPL backends, improving program validity and sampling quality beyond PyMC.

Table 9: Diagnostics and f-LOO results for NumPyro using Qwen-2.5-Coder-7B, demonstrating the generalizability of REFINESTAT.

Dataset	Variant	Reliab. Scr. \uparrow		$\widehat{R} \downarrow$		ESS Bulk \uparrow		Diverg. \downarrow		Pareto $k \downarrow$		ELPD LOO \uparrow	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
8 Schools	Standard	7.00	0.00	1.00	0.00	1769.00	0.00	0.00	0.00	0.13	0.00	-31.83	0.00
	REFINESTAT	7.00	0.00	1.00	0.00	1850.00	50.00	0.00	0.00	0.11	0.02	-31.10	0.05
Dugongs	Standard	7.00	0.00	1.00	0.00	1739.00	0.00	0.00	0.00	0.04	0.00	8.18	0.00
	REFINESTAT	7.00	0.00	1.00	0.00	1747.00	11.55	0.00	0.00	0.04	0.00	8.19	0.01
Peregrine	Standard	6.00	0.00	1.00	0.00	3232.50	81.32	0.00	0.00	0.00	0.00	-368.21	0.04
	REFINESTAT	6.67	0.47	1.00	0.00	1658.00	135.60	0.00	0.00	0.11	0.10	-76.25	122.44
Surgical	Standard	6.00	0.00	1.00	0.00	1472.00	0.00	0.00	0.00	0.25	0.00	-245.14	0.00
	REFINESTAT	7.00	0.00	1.00	0.00	1628.00	0.00	0.00	0.00	0.08	0.00	-50.37	0.00
GP	Standard	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
	REFINESTAT	6.67	0.47	1.00	0.00	2344.33	115.46	0.00	0.00	0.09	0.13	-83.07	82.70

Chapter 8 Related Work

Researchers have presented probabilistic techniques for discovering model structures from observed data, including Bayesian networks (Mansinghka et al., 2006; Lowd and Domingos, 2012), matrix-composition models (Roger B. Grosse et al., 2012), Markov networks (Gogate, Webb, and Domingos, 2010), and deep probabilistic models (Gens and Domingos, 2013). Probabilistic programming languages (PPLs) flexibly represent these models as programs but demand substantial domain and API expertise. Recent advances in LLM code generation make synthesizing probabilistic programs more accessible.

REFINESTAT leverages this opportunity to advance the state of the art in LLM-based probabilistic program synthesis.

Probabilistic Program Synthesis: There have been many approaches for deterministic program synthesis, which is recently been dominated by LLM-based approaches. Several works (Nori et al., 2015; Saad et al., 2019; Gerasimou, Tamburrelli, and Calinescu, 2015; Češka et al., 2019; Ellis, Solar-Lezama, and Tenenbaum, 2015) synthesize probabilistic programs using classical machine learning or symbolic methods. Prior work has also proposed techniques for debugging probabilistic programs (Dutta, W. Zhang, et al., 2019; Dutta, Huang, and Misailovic, 2022; Nussbaumer, Böck, and Cito, 2026), which could provide further feedback for probabilistic program synthesis. (Gerasimou, Tamburrelli, and Calinescu, 2015) uses a genetic algorithm for probabilistic model generation. (Saad et al., 2019) presents techniques to automatically construct probabilistic programs using Bayesian inference over DSLs defined via probabilistic grammars, enabling qualitative structure discovery and quantitative prediction.

Most recently, (M. Y. Li, Fox, and Goodman, 2024) uses LLM for probabilistic program synthesis. The paper shows that with instances of GPT4 as the generator and critic (closed LLM) can find reasonable probabilistic programs from data. As our evaluation shows, REFINESTAT significantly improves the ability to find programs that fits the data (recall Table 4) and in contrast to (M. Y. Li, Fox, and Goodman, 2024), runs only a single

small open LLM ($< 8B$ weights), demonstrating the benefits of constrained decoding.

Program Synthesis with Constrained LLM Decoding: Recent advances in program synthesis have enabled constrained decoding approaches where LLMs generate code while adhering to formal language specifications. These constraints can be partially precomputed and enforced more efficiently for regular (Deutsch, Upadhyay, and Roth, 2019; Willard and Louf, 2023; Kuchnik, Smith, and Amvrosiadis, 2023) or context-free (Koo, F. Liu, and He, 2024; Ugare, Suresh, et al., 2024; Yixin Dong et al., 2024; Banerjee et al., 2025; Suresh et al., 2025; Firestone et al., 2025) languages, ensuring syntactic correctness. Recent grammar-constrained and type-constrained approaches rely on a prefix property (Mündler et al., 2025), where each partial program can be incrementally validated and completed into a syntactic and/or well-typed program. In contrast, REFINESTAT targets properties that cannot be prefix-checked, since statistical reliability requires full posterior inference rather than purely static validation. For more dynamic program generation, Poesia et al. (2022) and Ugare, Gumaste, et al. (2025) implement error-driven backtracking.

Recently, several works have explored probabilistic inference/programming in LM-constrained generation. Loula et al. (2025) guide generation with potential scores and grammar rules, constraining token emission but not model correctness. In contrast, REFINESTAT generates probabilistic programs, enforces PPL checks during decoding, and retains only those passing Bayesian diagnostics. In Grand et al. (2025), a Planner writes an inference plan that LMs execute to satisfy constraints. REFINESTAT instead directly writes the probabilistic model and focuses on the quality of the posterior. Ahmed et al. (2025) adjusts next-token probabilities using a verifier so text matches high-level attributes. REFINESTAT aims for statistical validity, enforcing PPL semantics, and selecting the final program.

Chapter 9 Conclusions

The main contribution of our work is to separate the task of generating probabilistic modeling through PPLs as fragments of priors and likelihood and to construct an LLM-based search procedure that automatically discovers the probabilistic program that satisfies the standard reliability metrics in the Bayesian workflow. We believe that our framework can be extended to enforce arbitrary reliability criteria defined by domain experts for reliable generation in other domains that involve domain-specific languages and plan to explore those in future work.

Importantly, this approach is not specific to probabilistic programming. The core idea of structured generation with feedback-driven refinement can be extended to other programming language interfaces and domain-specific languages, where correctness and reliability are governed by domain-dependent constraints. We believe such a framework could enable smaller models to achieve strong performance by leveraging structured feedback, making it particularly relevant in settings where large closed models are impractical due to cost, privacy, or deployment constraints. Looking ahead, even as model capabilities improve, we expect this iterative loop of generation and validation to remain essential for complex reasoning tasks, where correctness cannot be guaranteed through scaling alone. We further envision extending our framework to incorporate richer and more targeted forms of feedback: while our current refinement strategy relies on heuristic signals (e.g., deciding whether to resample the prior or likelihood), future systems could identify and localize failure modes to specific components of the program and guide fine-grained modifications accordingly. Such directed feedback, combined with domain-expert-defined reliability criteria, could significantly improve the efficiency, interpretability, and generality of the search process across a wide range of domains beyond PPLs.

While our framework incorporates key components of the Bayesian workflow (i.e., convergence diagnostics and predictive performance metrics), it does not include prior-predictive or posterior-predictive checks, which often require manual inspection and

domain-specific judgment (Gelman, Carlin, et al., 1995). Thus, the reliability judgment is based on a subset of available diagnostics, and the reported ELPD only partially reflect model adequacy in some cases. Further, our refinement strategy is effective in practice but does not guarantee convergence to globally optimal program.

References

- Ahmed, Kareem, Catarina G Belem, Padhraic Smyth, and Sameer Singh (2025). *Semantic Probabilistic Control of Language Models*. arXiv: 2505.01954 [cs.LG]. URL: <https://arxiv.org/abs/2505.01954>.
- Banerjee, Debangshu, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh (2025). *CRANE: Reasoning with constrained LLM generation*. arXiv: 2502.09061 [cs.PL]. URL: <https://arxiv.org/abs/2502.09061>.
- Belcak, Peter, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov (2025). “Small language models are the future of agentic ai”. In: *arXiv preprint arXiv:2506.02153*.
- Betancourt, Michael (2017). “A conceptual introduction to Hamiltonian Monte Carlo”. In: *arXiv preprint arXiv:1701.02434*.
- Bingham, Eli, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman (2019). “Pyro: Deep universal probabilistic programming”. In: *Journal of machine learning research* 20.28, pp. 1–6.
- Bongard, Josh and Hod Lipson (2007). “Automated reverse engineering of nonlinear dynamical systems”. In: *Proceedings of the National Academy of Sciences* 104.24, pp. 9943–9948.
- Carpenter, Bob, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell (2017a). “Stan: A probabilistic programming language”. In: *Journal of statistical software* 76, pp. 1–32.
- (2017b). “Stan: A Probabilistic Programming Language”. In: *Journal of Statistical Software* 76.1, pp. 1–32. URL: <https://www.jstatsoft.org/index.php/jss/article/view/v076i01>.

- Češka, Milan, Christian Hensel, Sebastian Junges, and Joost-Pieter Katoen (2019). *Counterexample-Driven Synthesis for Probabilistic Program Sketches*. arXiv: 1904.12371 [cs.SE]. URL: <https://arxiv.org/abs/1904.12371>.
- Deutsch, Daniel, Shyam Upadhyay, and Dan Roth (2019). “A General-Purpose Algorithm for Constrained Sequential Inference”. In: *Proceedings of the Conference on Computational Natural Language Learning*. URL: <https://aclanthology.org/K19-1045/>.
- Dong, Yihong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li (2024). *Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models*. arXiv: 2402.15938 [cs.CL]. URL: <https://arxiv.org/abs/2402.15938>.
- Dong, Yixin, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen (2024). “XGrammar: Flexible and efficient structured generation engine for large language models”. In: *arXiv preprint arXiv:2411.15100*. URL: <https://arxiv.org/pdf/2411.15100>.
- Dutta, Saikat, Zixin Huang, and Sasa Misailovic (2022). “Sixthsense: Debugging convergence problems in probabilistic programs via program representation learning”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, pp. 123–144.
- Dutta, Saikat, Wenxian Zhang, Zixin Huang, and Sasa Misailovic (2019). “Storm: program reduction for testing and debugging probabilistic programming systems”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 729–739.
- Duvenaud, David, James Lloyd, Roger Grosse, Joshua Tenenbaum, and Ghahramani Zoubin (2013). “Structure discovery in nonparametric regression through compositional kernel search”. In: *International Conference on Machine Learning*. PMLR, pp. 1166–1174.

- Ellis, Kevin, Armando Solar-Lezama, and Josh Tenenbaum (2015). “Unsupervised Learning by Program Synthesis”. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28.
- Firestone, Preston, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic (2025). “UTF-8 Plumbing: Byte-level Tokenizers Unavoidably Enable LLMs to Generate Ill-formed UTF-8”. In: *Second Conference on Language Modeling*. URL: <https://openreview.net/forum?id=8ExXncFpf6>.
- Gelman, Andrew, John B Carlin, Hal S Stern, and Donald B Rubin (1995). *Bayesian data analysis*.
- Gelman, Andrew, Aki Vehtari, Daniel Simpson, Charles C Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák (2020). “Bayesian workflow”. In: *arXiv preprint arXiv:2011.01808*.
- Gens, Robert and Pedro Domingos (2013). “Learning the structure of sum-product networks”. In: *International conference on machine learning*. PMLR, pp. 873–880.
- Gerasimou, Simos, Giordano Tamburrelli, and Radu Calinescu (2015). “Search-based synthesis of probabilistic models for quality-of-service software engineering”. In: *30th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’15, pp. 319–330. URL: <https://doi.org/10.1109/ASE.2015.22>.
- Gogate, Vibhav, William Webb, and Pedro Domingos (2010). “Learning Efficient Markov Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 23.
- Golchin, Shahriar and Mihai Surdeanu (2024). *Time Travel in LLMs: Tracing Data Contamination in Large Language Models*. arXiv: 2308.08493 [cs.CL]. URL: <https://arxiv.org/abs/2308.08493>.
- (2025). *Data Contamination Quiz: A Tool to Detect and Estimate Contamination in Large Language Models*. arXiv: 2311.06233 [cs.CL]. URL: <https://arxiv.org/abs/2311.06233>.

- Gordon, Andrew D., Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani (2014). “Probabilistic Programming”. English. In: *Proceedings of the on Future of Software Engineering*, pp. 167–181.
- Grand, Gabriel, Joshua B. Tenenbaum, Vikash K. Mansinghka, Alexander K. Lew, and Jacob Andreas (2025). *Self-Steering Language Models*. arXiv: 2504.07081 [cs.CL]. URL: <https://arxiv.org/abs/2504.07081>.
- Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. (2024). “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783*.
- Grosse, Roger B., Ruslan Salakhutdinov, William T. Freeman, and Joshua B. Tenenbaum (2012). “Exploiting compositionality to explore a large space of model structures”. In: *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*. UAI’12. Catalina Island, CA, pp. 306–315. ISBN: 9780974903989.
- Grosse, Roger Baker (2014). “Model selection in compositional spaces”. PhD thesis. Massachusetts Institute of Technology.
- Guo, Daya, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. (2025). “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning”. In: *arXiv preprint arXiv:2501.12948*.
- Hoffman, Matthew D, Andrew Gelman, et al. (2014). “The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo.” In: *J. Mach. Learn. Res.* 15.1, pp. 1593–1623.
- Hui, Binyuan, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. (2024). “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186*.

- Kong, Jiaolong, Xiaofei Xie, and Shangqing Liu (2025). “Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework”. In: *Proceedings of the ACM on Software Engineering* 2.FSE, pp. 2712–2734.
- Koo, Terry, Frederick Liu, and Luheng He (2024). “Automata-based constraints for language model decoding”. In: *Conference on Language Modeling*. URL: <https://openreview.net/forum?id=BDBdblmyzY>.
- Kuchnik, Michael, Virginia Smith, and George Amvrosiadis (2023). “Validating large language models with RELM”. In: *Proceedings of Machine Learning and Systems* 5. URL: https://proceedings.mlsys.org/paper_files/paper/2023/file/93c7d9da61ccb2a60ac047e92787c3ef-Paper-mlsys2023.pdf.
- Li, Michael Y., Emily B. Fox, and Noah D. Goodman (2024). *Automated Statistical Model Discovery with Language Models*. arXiv: 2402.17879 [cs.LG]. URL: <https://arxiv.org/abs/2402.17879>.
- Li, Yucheng (2023). *Estimating Contamination via Perplexity: Quantifying Memorisation in Language Model Evaluation*. arXiv: 2309.10677 [cs.CL]. URL: <https://arxiv.org/abs/2309.10677>.
- Linka, Kevin, Sarah R St Pierre, and Ellen Kuhl (2023). “Automated model discovery for human brain using constitutive artificial neural networks”. In: *Acta Biomaterialia* 160, pp. 134–151.
- Loula, João, 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 (2025). *Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo*. arXiv: 2504.13139 [cs.CL]. URL: <https://arxiv.org/abs/2504.13139>.
- Lowd, Daniel and Pedro Domingos (2012). *Learning Arithmetic Circuits*. arXiv: 1206.3271 [cs.AI]. URL: <https://arxiv.org/abs/1206.3271>.
- M Kery, M Schaub (2011). *Bayesian population analysis using WinBUGS*.

- Magnusson, Måns, Jakob Torgander, Paul-Christian Bürkner, Lu Zhang, Bob Carpenter, and Aki Vehtari (2024). *posteriordb: Testing, Benchmarking and Developing Bayesian Inference Algorithms*. arXiv: 2407.04967 [stat.CO]. URL: <https://arxiv.org/abs/2407.04967>.
- Mansinghka, V. K., C. Kemp, J. B. Tenenbaum, and T. L. Griffiths (2006). “Structured priors for structure learning”. In: *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*. UAI’06. Cambridge, MA, USA, pp. 324–331. ISBN: 0974903922.
- McKinney, BA, JE Crowe Jr, HU Voss, PS Crooke, N Barney, and JH Moore (2006). “Hybrid grammar-based approach to nonlinear dynamical system identification from biological time series”. In: *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 73.2, p. 021912.
- Meent, Jan-Willem van de, Brooks Paige, Hongseok Yang, and Frank Wood (2021). *An Introduction to Probabilistic Programming*. arXiv: 1809.10756 [stat.ML]. URL: <https://arxiv.org/abs/1809.10756>.
- Mündler, Niels, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev (June 2025). “Type-Constrained Code Generation with Language Models”. In: *Proceedings of the ACM on Programming Languages* 9.PLDI, pp. 601–626. URL: <http://dx.doi.org/10.1145/3729274>.
- Neal, Radford M et al. (2011). “MCMC using Hamiltonian dynamics”. In: *Handbook of markov chain monte carlo* 2.11, p. 2.
- Nori, Aditya V., Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy (2015). “Efficient synthesis of probabilistic programs”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. Portland, OR, USA, pp. 208–217. ISBN: 9781450334686. URL: <https://doi.org/10.1145/2737924.2737982>.

- Nussbaumer, Nathanael, Markus Böck, and Jürgen Cito (2026). “Online and Interactive Bayesian Inference Debugging”. In: *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering*. ICSE '26. Rio de Janeiro, Brazil, p. 12.
- Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*.
- Phan, Du, Neeraj Pradhan, and Martin Jankowiak (2019). *Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro*. arXiv: 1912.11554 [stat.ML]. URL: <https://arxiv.org/abs/1912.11554>.
- Poesia, Gabriel, Alex 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*. URL: <https://openreview.net/forum?id=KmtVD97J43e>.
- Rubin, Donald B (1981). “Estimation in parallel randomized experiments”. In: *Journal of Educational Statistics* 6.4, pp. 377–401.
- Saad, Feras A., Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka (Jan. 2019). “Bayesian synthesis of probabilistic programs for automatic data modeling”. In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–32. URL: <http://dx.doi.org/10.1145/3290350>.
- Salvatier, John, Thomas V. Wiecki, and Christopher Fonnesbeck (Apr. 2016). “Probabilistic programming in Python using PyMC3”. In: *PeerJ Computer Science* 2, e55. URL: <https://doi.org/10.7717/peerj-cs.55>.
- Schmidt, Michael and Hod Lipson (2009). “Distilling free-form natural laws from experimental data”. In: *science* 324.5923, pp. 81–85.

- Suresh, Tarun, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh (2025). *DINGO: Constrained Inference for Diffusion LLMs*. arXiv: 2505.23061 [cs.LG]. URL: <https://arxiv.org/abs/2505.23061>.
- Team, CodeGemma, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. (2024). “Codegemma: Open code models based on gemma”. In: *arXiv preprint arXiv:2406.11409*.
- Ugare, Shubham, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic (2025). “IterGen: Iterative Structured LLM Generation”. In: *The Thirteenth International Conference on Learning Representations*. URL: <https://openreview.net/pdf?id=ac93gRzxxV>.
- Ugare, Shubham, Tarun Suresh, Hango Kang, Sasa Misailovic, and Gagandeep Singh (2024). *SynCode: LLM Generation with Grammar Augmentation*. arXiv: 2403.01632 [cs.LG]. URL: <https://arxiv.org/abs/2403.01632>.
- Unit, MRC Biostatistics (n.d.[a]). *Examples Volume 1*. URL: http://www.mrc-bsu.cam.ac.uk/wp-content/uploads/WinBUGS_Vol1.pdf.
- (n.d.[b]). *Examples Volume 2*. URL: http://www.mrc-bsu.cam.ac.uk/wp-content/uploads/WinBUGS_Vol2.pdf.
- Vehtari, Aki, Andrew Gelman, and Jonah Gabry (2017). “Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC”. In: *Statistics and computing* 27, pp. 1413–1432.
- Vehtari, Aki, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner (2021). “Rank-normalization, folding, and localization: An improved \hat{R} for assessing convergence of MCMC (with discussion)”. In: *Bayesian analysis* 16.2, pp. 667–718.

Willard, Brandon T and Rémi Louf (2023). “Efficient guided generation for large language models”. In: *arXiv preprint arXiv:2307.09702*. URL: <https://arxiv.org/pdf/2307.09702>.