# ALPS: CLIMBING THE PROBABILISTIC PROGRAM LANDSCAPE WITH LLM-GUIDED SYNTHESIS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Trustworthy modeling requires more than accurate predictions: it requires confidence estimates grounded in an explicit understanding of the process that generated the data. Models written as probabilistic programs (PPs) meet this need directly, as each program represents a transparent and human-readable generative process, with dependencies and noise sources laid out as executable code. Several approaches have explored constructing probabilistic programs from data alone, without prior knowledge of the data-generating process, but the problem remains challenging, as the space of possible structures is combinatorially large, and parameter optimization requires a fixed structure. We introduce ALPS (Automated LLM-guided Probabilistic programs Synthesis), a pipeline that pairs LLM-guided structural proposal and mutation with gradient-based optimization of each candidate's continuous parameters. This approach combines informed and diagnostics-driven structural search with efficient gradient-based parameter refinement. Across nine benchmarks common in the PP literature, we compare ALPS against four baselines and show that our approach attains the best likelihood fit on average on all the benchmarks and remains competitive in distributional distance to the ground-truth program. Beyond these metrics, we show qualitatively that ALPS more reliably recovers the correct correlation structure underlying the data, yielding programs that are not only accurate but also faithful to the underlying generative process. We further validate ALPS on a real-world dataset, and show that prior domain knowledge can be injected as natural-language hypotheses.

## 1 INTRODUCTION

Probabilistic Programs (PPs) offer a compact and interpretable way to specify generative models of data, combining the expressiveness of general-purpose programming languages with the semantics of probability distributions. This expressiveness, however, comes at a cost: writing a PP that matches the structural properties of a dataset (e.g. conditional dependencies between variables) typically requires full understanding of the process and expertise in probabilistic programming. Instead, usually, in real-case scenarios, only observations are available, i.e. observational datasets, with the processes behind the data hidden to the observer. Learning the data generating process in the form of a PP, i.e. PP synthesis, has therefore become an active area of research, with the goal of automatically discovering a program whose structure and parameters jointly explain a given dataset.

Existing approaches to PP synthesis explore program structures using syntax-aware but semantically blind variation operators, or enforce grammar validity directly during generation. Evolutionary and MCMC-based methods apply mutations and crossovers according to fixed rules, with no understanding of which edits are likely to improve fit, so they must rely on large populations or long sampling chains, using selection pressure — not informed search — to separate productive changes from unproductive ones. Constrained-decoding approaches instead enforce grammar validity directly during generation, but the models used are typically small and not conditioned on optimization feedback, limiting their ability to propose structurally meaningful edits.

Large language models (LLMs) offer an opportunity to close this gap: trained on large code and natural-language corpora, they encode strong priors over plausible program structures, and, placed in an iterative refinement loop, can condition proposals on structured feedback such as per-candidate loss deltas and parameter diagnostics — turning mutation from a blind stochastic operator into



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

one that approximates an informed local search direction, without requiring the search space to be differentiable.

In this paper, we introduce ALPS (Automated LLM-guided Probabilistic programs Synthesis), a system that couples LLM-driven structural search with gradient-based optimization of the relevant continuous parameters of the candidate PPs. In order to allow the computation of gradients in PPs, we resort to DeGAS, a constrained domain-specific language for PPs that admits the computation of gradients with respect to model parameters and an analytic approximation of the likelihoods. ALPS pipeline works through four stages: an LLM Module proposes and mutates program structures conditioned on optimization feedback, DeGAS evaluation scores candidates by log-likelihood, Gradient Refinement optimizes continuous parameters via automatic differentiation, and a selection step retains the top-$M$ programs for the next generation.

Our contributions are as follows: 1) we introduce ALPS, a PP synthesis system that replaces the blind variation operators typical of evolutionary search with an LLM-driven structural search conditioned on optimization diagnostics, coupled with gradient-based parameter optimization; 2) we evaluate ALPS against four baselines on nine benchmarks common in the PP literature, each defined by a known ground-truth PP that serves both as the data-generating process and as the reference for evaluating distributional recovery, and show that it achieves the best likelihood fit on average on all the benchmarks and remains competitive in distributional distance to the ground-truth program; we further show qualitatively on three representative benchmarks that ALPS recovers the correct dependency structure between variables; 3) through ablations, we show that ALPS's gains are sensitive to the scale of the underlying LLM backend, and that gradient-based parameter refinement contributes a substantial share of the improvement over structural search alone; and 4) we test ALPS on a real-world dataset and compare the results with a state-of-the-art method, further showing that prior domain knowledge can be injected directly as natural-language hypotheses, requiring no probabilistic-programming expertise from the user.

The remainder of this paper is organized as follows. Sec. 2 reviews background on PP synthesis, LLM-guided synthesis, and DeGAS, and it situates ALPS with respect to prior work. Sec. 3 describes the ALPS pipeline in detail, while Sec. 4 and Sec. 5 present our experimental evaluation and results.

## 2 Background & Related Work

### 2.1 Synthesis of PPs

Probabilistic Programming languages (PPLs) combine standard programming constructs with primitives for sampling and conditioning, making it possible to express complex generative models compactly and to reason about them via automated inference (Gordon et al., 2014). However, writing a PP by hand requires substantial prior knowledge of the data-generating process, which is often unavailable. This motivates the need for methods that can automatically synthesize a PP directly from a dataset of observations, without assuming prior knowledge of how the data was generated.

A pioneering approach to this problem is PSKETCH (Nori et al., 2015), that adapts the sketching paradigm from deterministic program synthesis: a user supplies a program skeleton with unspecified holes, and MCMC search instantiates them, approximating likelihoods with mixtures of Gaussians. Saad et al. (2019) instead cast synthesis as Bayesian inference over probabilistic context-free grammars, producing a posterior ensemble of programs. While this captures uncertainty over model structure, it sacrifices the transparency of a single, human-readable program. Gerasimou et al. (2015) explore PP synthesis for quality-of-service and software-reliability settings, but are limited to finite-state models verified against formal specifications rather than general fit to data. Closest to our setting, Doz et al. (2025) formulate PP synthesis as grammatical evolution, exploring the PP landscape via likelihood-guided evolutionary search with optional sketching. This approach relies on generic, semantics-agnostic mutation and crossover operators that manipulate program structure without exploiting gradient information: while gradients could in principle be used to tune numerical parameters within a fixed program structure, they are not preserved across generations, as structural operators act on the program syntax and have no mechanism to propagate parameter gradients to offspring. This motivates replacing the evolutionary operators with an LLM, which can leverage learned semantic knowledge to propose more informed structural modifications.

In general, LLMs' code-generation and domain-knowledge priors have made them attractive replacements for hand-crafted search operators or grammars. Li et al. (2024)'s BoxLM iteratively proposes and critiques PPs via an LLM acting as modeler and domain expert. Kanda et al. (2025) address the tendency of small LLMs to generate invalid programs via semantic constraints and a diagnostic-aware refinement loop. Domke (2025) focuses on generating many candidate models from an informal description and combining their predictions via importance weighting — a different objective from ALPS's search for a single transparent explanatory model. Wong et al. (2025) also use an LLM to synthesize PPs, but to support open-world reasoning about novel situations described in natural language, rather than to find a single best-fit model for a fixed dataset.

Beyond PPs, LLMs have driven broader scientific model discovery: Agarwal et al. (2025)'s AutoDiscovery uses an LLM's Bayesian surprise as a reward signal within Monte Carlo tree search, while Wahl et al. (2026)'s ModelSMC formalizes LLM-based model discovery as Sequential Monte Carlo, with candidate models as particles proposed, refined, and reweighted by likelihood.

Across these approaches, the LLM typically proposes and revises program structure while a separate mechanism scores candidates. ALPS follows this division of labor but couples LLM-driven structural proposals with gradient-based optimization.

### 2.2 DeGAS

DeGAS (Randone et al., 2026) introduces a differentiable semantic framework for loop-free PPs, enabling analytical likelihood approximation and gradient-based parameter optimization. It builds on SOGA (Randone et al., 2023), which represents each program variable's distribution as a Gaussian Mixture (GM) and propagates the distributions analytically through affine transformations, products, and truncation-based conditioning, yielding closed-form posteriors. DeGAS extends SOGA with a differentiable semantics that handles branching and conditioning discontinuities via a vanishing smoothing scheme, converging to unsmoothed SOGA semantics as the smoothing parameter tends to zero. Because posteriors and path probabilities are differentiable in program parameters, DeGAS supports end-to-end optimization via automatic differentiation. Moreover, since the posterior is approximated by a GM, the log-likelihood of the observed data $\mathbf{x}$ under the model admits a closed-form expression:

$$\log p(\mathbf{x}) = \log \sum_{k=1}^{K} \pi_k \, \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{1}$$

where $\pi_k > 0$ are the mixture weights with $\sum_{k=1}^{K} \pi_k = 1$, $\boldsymbol{\mu}_k \in \mathbb{R}^d$ is the mean vector, and $\boldsymbol{\Sigma}_k \in \mathbb{R}^{d \times d}$ is the covariance matrix of the $k$-th component. We describe the DeGAS syntax to the LLM through a prompt that presents the grammar in compact form and warns against common mistakes. Below, we illustrate the DeGAS syntax and semantics through a running example, which we also use as a benchmark in our experiments (see Table 3); the detailed grammar is given in App. A.1.

**Running example.** The program *If* first draws `a` from a Gaussian with mean 1 and variance 2. The subsequent `if` statement branches on the predicate `a < 0`, which partitions the support of `a` into two regions.

```
a = gm([1],[1.],[2.]);
if a < 0 {
    b = 3 * a + gm([1],[0.],[1.]);
} else {
    b = gm([1],[8.],[1.]);
} end if;
```

Within each branch, `b` is then defined conditionally on this truncated distribution of `a`: in the `if` branch, `b` is an affine function of `a` plus independent Gaussian noise, so the joint distribution of (`a`, `b`) is correlated; in the `else` branch, `b` is drawn independently of `a`.

At the end of the `if` statement, DeGAS merges the two branches by summing their contributions, each weighted by the probability mass assigned to that branch — that is, the overall distribution over (`a`, `b`) is a GM in which each branch contributes to its own components, scaled by the probability of having taken that branch.

# 3 METHOD

We introduce ALPS (Automated LLM-guided Probabilistic programs Synthesis), a system that couples LLM-driven structural search with DeGAS, a PPL that admits analytical log-likelihood approximation and gradient-based optimization of the program parameters. The pipeline mirrors the structure of an evolutionary algorithm: a population of candidate DeGAS programs is scored by fitness (log-likelihood), the fittest are retained, and semantically-aware variation operators produce the next generation, until a stopping criterion is met. What distinguishes ALPS is the nature of its variation operator. Classical genetic programming applies syntax-aware but semantically blind edits, relying on population-wide selection over many generations to separate beneficial variations from destructive ones. A similar shift from blind to LLM-guided variation operators has recently been explored in evolutionary computation more broadly, for algorithm and metaheuristic design (van Stein & Bäck, 2024; Viktorin et al., 2025). Figure 1 and Algorithm 1 summarize the full ALPS pipeline, whose stages we describe below.

**Input:** Data $\mathcal{D}$, templates $\{\tau_k\}_{k=1}^{K}$, grammar $\mathcal{G}$, hyperparams: $M, i, j, T, p, \varepsilon$
**Output:** $P^* = \arg\max_{P \in pool} \ell(P)$

1: $pool \leftarrow \emptyset$, $t \leftarrow 0$
2: $dict \leftarrow \text{LLMGENERATION}(stats(\mathcal{D}), \mathcal{G}, \{\tau_k\}_{k=1}^{K})$ *// LLM Generation*
3: $cands \leftarrow \text{PARSEJSON}(dict)$
4: **while** $t < T$ **do**
5: **for** $P \in cands$ **do** *// Evaluate + refine*
6: $\ell(P) \leftarrow \text{LOGLIKELIHOOD}(P, \mathcal{D})$
7: $P, \ell(P) \leftarrow \text{GRADREFINE}(P, \mathcal{D}, i)$
8: **end for**
9: $pool \leftarrow \text{TOP-}M(pool \cup cands)$ *// Pool update*
10: **for** $P \in pool$ **do** *// Refine and keep gradient info*
11: $P, \ell(P), g(P) \leftarrow \text{GRADREFINE}(P, \mathcal{D}, j)$
12: **end for**
13: **if** $\text{CONVERGE}(pool, p, \varepsilon)$ **then break**
14: **end if**
15: $dict \leftarrow \text{LLMMUTATE}(pool, g(pool), \mathcal{G})$ *// LLM Mutation*
16: $cands \leftarrow \text{PARSEJSON}(dict)$
17: $t \leftarrow t + 1$
18: **end while**
19: **return** $\arg\max_{P \in pool} \ell(P)$

Algorithm 1: Pseudocode of ALPS pipeline

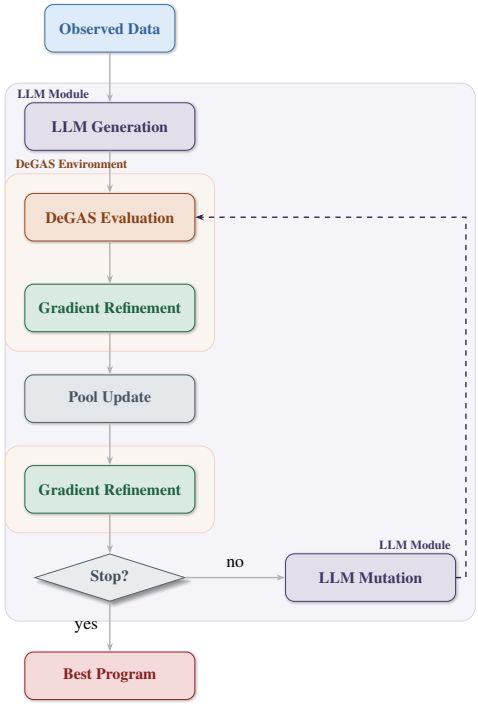



Figure 1: Overview of ALPS pipeline.

## 3.1 ALPS PIPELINE

**LLM Generation (lines 2-3).** In this step, LLMGENERATION takes in input a prompt with summary statistics of the dataset $stats(\mathcal{D})$, the DeGAS grammar specification $\mathcal{G}$ in compact form, and a set of structural templates $\{\tau_k\}_{k=1}^{K}$ that can guide the LLM towards diversity across the candidates by encoding different modeling assumptions as, for example, a single Gaussian-mixture component, or a latent variable with if/else branching. A template can also encode prior domain knowledge, as an explicit dependency hypothesis, letting a practitioner seed the initial candidate pool with structures they suspect are present in the data. The full generation prompt is reported in App. A.1.

Every LLM call requests a fixed JSON schema via structured-output decoding, ensuring that responses conform to the expected program format. PARSEJSON handles parsing failures through a corrective retry, and, if that also fails, a salvage step that tries to recover valid candidates from the malformed output. Only if both fail is the pool discarded; full details are given in App. A.3.

The output of this step is a set of $K$ candidate PPs, each generated with reference to one such templates.

**Evaluation and parametric optimization (lines 5-8).** In this phase, the input is the set of $K$ candidate PPs, which enter in the *DeGAS Environment* where they are compiled and their log-likelihood approximation is computed via DeGAS inference (LOGLIKELIHOOD$(P, \mathcal{D})$).

Notice that, unlike other approaches such as Kanda et al. (2025), validity is not enforced as a hard constraint during generation. We rely instead on post-hoc filtering, and any candidate raising a compilation exception at this step is discarded. GRADREFINE$(P, \mathcal{D}, i)$ exploits the resulting likelihood surface to optimize the means and standard deviations of the non-degenerate GM distributions declared in the PP for a fixed number of iterations $i$. Standard deviations are optimized in log-space ($\sigma = \exp(\rho)$) to enforce positivity and improve numerical stability. The output is a set of $K$ candidate PPs, each with its optimized parameters and log-likelihood evaluation.

**Pool update and optimization (lines 9-12).** The refined candidates ($cands$) are merged with the surviving population from the previous iteration ($pool$), and TOP-$M(pool \cup cands)$ retains the top-$M$ programs by log-likelihood. These PPs undergo a second GRADREFINE$(P, \mathcal{D}, j)$ pass with $j > i$ iterations: since a good program has already been aligned to the data by the first pass, a longer refinement is warranted to exploit this structure and push the parameters closer to their optimum.

**Convergence check (line 13).** CONVERGE$(pool, p, \varepsilon)$ takes in input a set of $M$ candidates with their log-likelihood evaluations, and determines whether to stop the algorithm: the loop terminates if the best loss in the pool has not improved by more than $\epsilon$ over the last $p$ iterations. This prevents unnecessary structural complexity from accumulating in candidates whose fit has already plateaued. If the criterion is not met, the pool is passed to the LLM mutation step, otherwise the highest-log-likelihood program (calculated on $\mathcal{D}$) in the pool is returned as the final output. Additionally, a maximum number of iterations $T$ is enforced by the *while* condition at line 4.

**LLM mutation (lines 15-16).** In this stage, LLMMUTATE$(pool, g(pool), \mathcal{G})$ takes in input the $M$ candidate PPs, and, providing a prompt (see App. A.1) that contains information about the gradient optimization of the candidates ($g(pool)$, i.e. how much the loss and parameters moved during gradient refinement) and DeGAS grammar $\mathcal{G}$, it asks the LLM to propose one mutated program for each candidate. Mutations are explicitly steered toward structural changes, since numeric tuning is already handled by gradient refinement. The prompt also suggests a set of illustrative operators as starting points, as splitting or merging mixture components, adding or removing a conditional branch, altering the dependency structure, or discarding a poor candidate in favor of a new hypothesis, though the LLM is free to propose any valid DeGAS structure.

The mutated candidates are again parsed by PARSEJSON and re-enter the DeGAS Environment at the DeGAS Evaluation stage (dashed arrow in Figure 1), closing the loop.

## 4 EXPERIMENTS

We evaluate ALPS on the nine benchmark PPs, drawn from Chasins & Phothilimthana (2017); Nori et al. (2015). The suite spans programs with two to four variables, covering Gaussian, Bernoulli, and categorical distributions, and includes both continuous and discrete variables (see App. C for more details). The scale of our benchmarks is consistent with established practice in data-driven PP synthesis from purely observational data (Nori et al., 2015; Kanda et al., 2025; Chasins & Phothilimthana, 2017), reflecting a shared constraint of the sub-field rather than a limitation specific to ALPS. Each target program $P^*$ generates a fixed training set of $n' = 1{,}000$ points, reused unchanged across every fitness evaluation of the search; the metrics are evaluated on a separate held-out set of $|\mathcal{D}_{\text{test}}| = 50{,}000$ points. The hyperparameters of all the pipeline are detailed in App. A.2. The code to reproduce all experiments is publicly available.[1] We compare ALPS against three classes of baselines: MCMC-based, grammatical evolution, and LLM-based search.

**MCMC baseline.** We consider a Monte Carlo-based approach to generating DeGAS PPs, similar in spirit to PSKETCH (Nori et al., 2015) but without requiring a sketch, since we aim to learn the entire program rather than fill in holes, and exploiting DeGAS's ability to compute exact likelihoods directly. MCMC search is performed only over the discrete program structure, while the continuous

[1] https://anonymous.4open.science/r/ALPS_PP

parameters of each proposed program are fit by gradient descent before scoring. Implementation details and hyperparameters are reported in App. C.1.

**Evolutionary baseline.** We use the grammatical evolution approach of Doz et al. (2025), with the hyperparameters reported in App. C.3. Unlike ALPS, this baseline relies solely on evolutionary operators for both structure and parameter search, with no gradient-based refinement.

**LLM-based baselines.** For the LLM-based baseline, we selected RefineStat (Kanda et al., 2025). However, since DeGAS does not expose a posterior suitable for MCMC sampling, RefineStat's refinement procedure and reliability checks could not be applied directly. We therefore implemented RS-MCMC, a pipeline that generates valid DeGAS programs, translates them into NumPyro to exploit RefineStat's MCMC engine, and retranslates the results back into DeGAS for metric computation. We note that this comparison is not directly with RefineStat but with our DeGAS adaptation.

To further compare against an LLM-based synthesis engine that also exploits gradients, we propose RS-Grad, a variant of RS-MCMC that replaces MCMC-based Bayesian inference with DeGAS's own gradient optimization. Full implementation details for both RS-MCMC and RS-Grad are reported in App. C.2. Notice that, in this work, RS-MCMC and RS-Grad use `Qwen2.5-Coder-7B-Instruct` LLM backend rather than the larger `gpt-oss:120b` used for ALPS. RefineStat's SynCode-based constrained decoding needs direct access to the model's raw per-token output distribution at every generation step, so that it can mask out grammar-violating tokens before sampling; this is only possible when the model is loaded directly into the same process (e.g. via HuggingFace transformers (Wolf et al., 2020)), not through a completion-style serving API such as Ollama's (Morgan & Chiang, 2026), which only returns already-sampled text and never exposes that distribution. Due to computational constraints, RefineStat's grammar-constrained decoding is restricted to smaller, locally-loadable models. We acknowledge that this limits the fairness of the direct comparison, and leave an evaluation with larger LLM backends to future work.

**Real-case scenario.** Separately, to also obtain a direct comparison point where DeGAS compatibility is not a constraint, we ran RefineStat exactly as released, generating and refining native PyMC programs rather than DeGAS ones, on a real-world dataset. We further show that ALPS allows prior domain knowledge to be injected directly in natural-language, requiring no probabilistic-programming expertise from the user, a capability no other baseline exposes.

**Ablation analysis.** We also conducted a series of ablation studies to assess the contribution of each component of ALPS. We analyzed the impact of the choice of LLM backbone, since ALPS relies on the LLM to propose structural modifications and interpret optimization diagnostics. We therefore substitute the generation backend across eight open-weight models spanning four families and roughly an order of magnitude in scale (7B–120B). We further ablate the gradient refinement step, the LLM mutation step, and the gradient information passed to the mutation prompt, to quantify the individual contribution of each component to the overall performance of the pipeline.

**Metrics.** We evaluate synthesized programs along two axes: predictive accuracy on held-out data, and closeness to the ground-truth output distribution. For the former, we compute the negative log-likelihood (NLL) of the synthesized program on a held-out test set $\mathcal{D}_{\text{test}}$, normalized by the number of data points. For the latter, we compare the output distribution of the synthesized program against that of the ground-truth program, both represented as GMs, using the GM Wasserstein-type distance $W_2^2$ proposed by Delon & Desolneux (2020). Note that, since synthesis is evaluated from observational data alone, the ground-truth program is not necessarily identifiable, as multiple programs may induce equivalent or near-equivalent distributions. Our distributional metrics therefore evaluate behavioral recovery rather than exact program recovery. For the real-world case study, no closed-form reference distribution is available to compute the $W_2^2$ distance, so we instead report the empirical Wasserstein-1 (W1) distance (Ramdas et al., 2015) between each method's samples and the real held-out data.

## 5 Results

**Comparison with baselines.** Table 1 reports NLL and $W_2^2$ on held-out data for ALPS and four baselines across all nine benchmarks. ALPS achieves the best NLL on average on every bench-

Table 1: NLL and Gaussian-Wasserstein distance ($W_2^2$) on held-out data, mean $\pm$ std over 5 independent runs per method. A cell reports *n.c.* (not converged) whenever 2 or more of its 5 runs failed to produce a valid candidate. Lower is better for both the metrics; **bold** marks the best value per benchmark/metric among reporting methods.

| Benchmark | Metric | ALPS | RS-MCMC | RS-Grad | MCMC | Evolut. |
|---|---|---|---|---|---|---|
| If | NLL | $\mathbf{4.01 \pm 0.06}$ | $4.77 \pm 0.40$ | $5.25 \pm 0.09$ | $5.41 \pm 0.12$ | $5.29 \pm 0.25$ |
| | $W_2^2$ | $\mathbf{8.31 \pm 9.03}$ | $16.40 \pm 14.55$ | $47.76 \pm 9.27$ | $282.1 \pm 295.6$ | $65.38 \pm 19.21$ |
| MoG | NLL | $\mathbf{6.65 \pm 0.13}$ | $6.72 \pm 0.01$ | $7.14 \pm 0.59$ | $7.56 \pm 0.49$ | $7.73 \pm 0.45$ |
| | $W_2^2$ | $16.20 \pm 5.54$ | $\mathbf{8.53 \pm 0.61}$ | $17.84 \pm 12.39$ | $476.3 \pm 869.5$ | $38.75 \pm 46.71$ |
| CSI | NLL | $\mathbf{-20.79 \pm 0.39}$ | $1.79 \pm 0.00$ | *n.c.* | $4.24 \pm 1.20$ | $-20.53 \pm 0.91$ |
| | $W_2^2$ | $1.02 \pm 0.18$ | $1.27 \pm 0.02$ | *n.c.* | $707.4 \pm 704.2$ | $\mathbf{0.43 \pm 0.14}$ |
| EasyTugWar | NLL | $\mathbf{0.23 \pm 0.13}$ | *n.c.* | $5.57 \pm 1.56$ | $7.17 \pm 0.41$ | $1.71 \pm 2.74$ |
| | $W_2^2$ | $\mathbf{12.61 \pm 4.51}$ | *n.c.* | $18.82 \pm 14.92$ | $234.9 \pm 299.3$ | $16.85 \pm 6.38$ |
| BiasedTugWar | NLL | $\mathbf{1.36 \pm 2.64}$ | *n.c.* | $5.88 \pm 1.51$ | $6.81 \pm 0.80$ | $8.79 \pm 14.04$ |
| | $W_2^2$ | $\mathbf{9.21 \pm 2.50}$ | *n.c.* | $54.40 \pm 92.73$ | $937 \pm 1380$ | $26341 \pm 58808$ |
| MixedCond. | NLL | $\mathbf{-0.87 \pm 0.04}$ | $5.04 \pm 0.00$ | $4.29 \pm 1.68$ | $6.17 \pm 0.51$ | $1.29 \pm 1.80$ |
| | $W_2^2$ | $3.43 \pm 0.50$ | $2.92 \pm 0.10$ | $\mathbf{2.87 \pm 0.12}$ | $447.4 \pm 599.2$ | $43.55 \pm 82.27$ |
| MultiBranch | NLL | $\mathbf{5.98 \pm 0.12}$ | $6.23 \pm 0.15$ | $6.30 \pm 0.20$ | $6.40 \pm 0.10$ | $6.46 \pm 0.21$ |
| | $W_2^2$ | $\mathbf{25.00 \pm 8.41}$ | $44.10 \pm 3.88$ | $61.27 \pm 18.77$ | $61.71 \pm 2.33$ | $93.58 \pm 35.64$ |
| EyeColor | NLL | $\mathbf{-13.21 \pm 0.51}$ | $3.76 \pm 0.01$ | $2.22 \pm 0.15$ | $3.86 \pm 0.75$ | *n.c.* |
| | $W_2^2$ | $4.21 \pm 0.80$ | $5.21 \pm 0.10$ | $\mathbf{3.47 \pm 1.55}$ | $1519 \pm 1791$ | *n.c.* |
| Hurricane | NLL | $\mathbf{-9.96 \pm 0.14}$ | $0.13 \pm 4.13$ | $0.01 \pm 0.11$ | $2.49 \pm 0.15$ | $-9.76 \pm 0.66$ |
| | $W_2^2$ | $0.35 \pm 0.07$ | $1.69 \pm 0.71$ | $0.78 \pm 0.01$ | $135.6 \pm 126.7$ | $\mathbf{0.34 \pm 0.10}$ |

mark and leads on $W_2^2$ on average for four of them, with no systematic gap between continuous, mixed, and discrete benchmarks. This advantage is statistically significant on NLL against every baseline except Evolutionary, where it holds only on non-discrete benchmarks; on $W_2^2$ it is significant against MCMC on most benchmarks, but more mixed against the other three, though ALPS is rarely significantly outperformed (App. D.1).

Notice that on benchmarks with discrete-valued variables, DeGAS encodes each point mass as a narrow Gaussian component of near-zero variance $\epsilon$; the resulting density (not probability) can exceed 1 and contributes an unbounded $\log\left(1/\sqrt{2\pi\epsilon}\right)$ bonus per discrete variable to the log-likelihood, which is why NLL can be negative on these benchmarks. Since every method we compare against compiles to this same DeGAS representation, all of them are equally capable of exhibiting this artifact, as it is not specific to any one method's search procedure, but to whether its synthesized programs happen to commit to exact point masses ($\sigma \to 0$) for discrete variables. In practice, RS-MCMC's and RS-Grad's synthesized programs rarely do: rather than a mixture with components located at the data's modes, they consistently fall back on a single wide Gaussian per variable, showing a genuine difficulty capturing multimodal and discrete structure – which is why ALPS and the evolutionary approach, whose candidates do commit to such mixtures, generally achieve better NLL and $W_2^2$ on the discrete benchmarks. Fig. 4 in App. D.2 shows graphically this effect on `csi` benchmark.

To complement the quantitative metrics, we assess whether the synthesized programs recover the correct dependency structure between variables. For three representative benchmarks — the continuous `if` example, shown in Figure 2, one mixed benchmark, and one discrete benchmark, both reported in App. D.2 — we draw samples from the best synthesized program and compare them against samples from the ground-truth program. For the running example, ALPS is the only method that recovers the correct dependency structure between $a$ and $b$: samples correctly separate into distinct regimes depending on the value of $a$, matching the branching structure of the true generative program, and the support of the synthesized distribution closely tracks the true data manifold. By contrast, the baselines fail to capture this conditional structure. RS-MCMC and RS-Grad partially recover the dense cluster for $a > 0$ but scatter samples broadly and place non-trivial mass outside the true support for $a < 0$. MCMC largely ignores the dependency between $a$ and $b$, producing diffuse clouds that occupy far more of the sample space than the true data supports. Evolutionary partially reconstructs the two-regime structure but with more noise within each regime than ALPS.

From the two benchmarks shown in the App. D.2, we can see that only ALPS and the Evolutionary approach succeed in capturing discrete behaviors of the data, and, on a mixed benchmark, the MCMC-based methods struggle to recover the separation between groups, while RS-Grad, Evolutionary, and ALPS all capture the overall structure, with ALPS generating samples closer to the data manifold. Notice that a method can post a competitive NLL or $W_2^2$ while still misrepresenting the underlying dependencies between variables. This distinction is important, particularly from the perspective of synthesizing causal models represented as PPs, as in Doz et al. (2026).

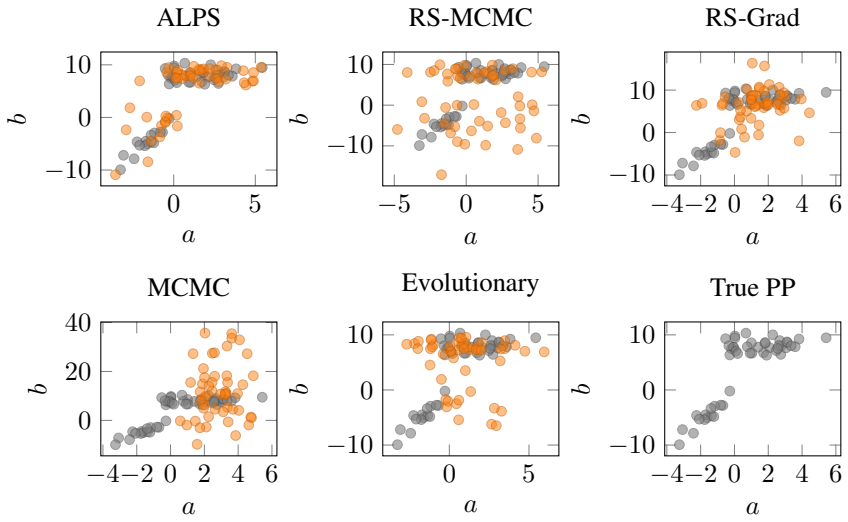



Figure 2: Comparison of samples drawn from each synthesized program (orange) against samples from the real program (gray), across all five methods on the running `If` example (best candidate out of 5 runs). ALPS recovers the correct dependency structure between $a$ and $b$ and tracks the true data manifold.

**Computational budget.** Methods differ substantially in nature (LLM calls, gradient-based refinement, or sampling), so their computational budgets are not directly comparable. Where a method has no natural convergence signal of its own, the iteration budget is set empirically to the point beyond which further compute no longer changes the outcome. For the RefineStat-based methods, we instead considered the budget reported in the original paper. Table 2 reports the resulting average wall-clock time per run, pooled across all nine benchmarks. App. D.7 details how each budget was fixed.

| Method | Time (min) |
|---|---|
| ALPS | $16.3 \pm 8.7$ |
| RS-Grad | $4.3 \pm 3.0$ |
| RS-MCMC | $237.2 \pm 246.4$ |
| MCMC | $48.8 \pm 21.2$ |
| Evolutionary | $10.8 \pm 18.0$ |

Table 2: Average wall-clock time per run, mean $\pm$ std. across all benchmarks.

**LLM backend scalability.** We evaluated ALPS across 8 LLM backends of varying scale and capability on the `if` benchmark (App. D.3). DeGAS compilation failure rates range from under 5% to 100% and model capability, not just scale, determines whether ALPS reliably produces a usable program. This is expected, as ALPS relies on the model following DeGAS's grammar from prompt instructions. Its generate-and-select loop can absorb moderate failure rates, but not the weakest backends' complete or near-complete failure.

**Ablation analysis.** We conduct three ablation studies to assess the contribution of each component of ALPS: we ablate gradient refinement (App. D.4), the gradient information passed to the mutation prompt (App. D.5), and LLM-guided mutation (App. D.6). Gradient refinement significantly improves NLL on 6 of 9 benchmarks (Mann-Whitney, $p < 0.05$), with no significant effect on `mog` (where the two configurations are nearly identical to begin with), `easytugwar`, or `biasedtugwar`. The effect on $W_2^2$ is more limited, reaching significance on only 3 of 9 benchmarks. Removing the gradient step yields a modest wall-clock speedup (18–32%), but run time remains dominated by LLM calls. Passing gradient information (loss and parameter deltas) to the

LLM mutation prompt has, at this sample size, no consistently demonstrable effect. This suggests the LLM can propose productive mutations from the program and data statistics alone, with gradient information contributing at most a small, benchmark-dependent effect. LLM-guided mutation substantially improves average NLL across all the benchmarks – statistically significant on 6 of 9 – with the largest gains on discrete and mixed benchmarks, confirming that iterative structural search is a primary driver of ALPS's performance.

**Case study: real-world scenario.** To test whether ALPS can recover meaningful structure from real (not synthetic) data, we ran it on a reduced 8-variable subset of the UCI Heart Disease dataset (Janosi & Detrano, 1989), and compare against the original RefineStat method. We ran ALPS with the same hyperparameters and prompt template as the synthetic benchmarks (App. A.2), obtaining 3 of 5 valid runs – the other two failed when the LLM substituted placeholder variable names for the real column names. RefineStat produced valid candidates on only 2 of 5 seeds; of these, one includes `num` among its own predictors, so the fitted model trivially reconstructs the target rather than learning it. We therefore evaluate $W_1$ on RefineStat's best candidate – which is the one selected by RefineStat's own reliability diagnostics – against ALPS's best candidate across its 3 converged seeds, reflecting the outcome a practitioner would obtain by running a few seeds and keeping the best one. We obtain $W_1 = 0.240$ for ALPS and $W_1 = 0.513$ for RefineStat. This difference is reflected in the marginal distribution of `num` (Fig. 3, left).

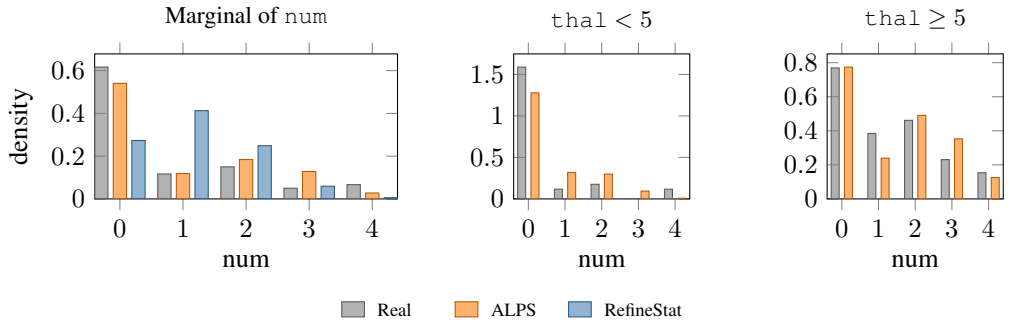



Figure 3: Left: marginal distribution of `num` (real vs. ALPS and RefineStat). Center and right: conditional distribution of `num` by `thal` branch, real vs. ALPS's best candidate.

To show that domain knowledge can be injected into the search without any probabilistic programming expertise, we reran this real-world case study replacing two of the five generic structural templates with plain-English hypotheses, e.g. *"num depends primarily on thal"*. The best resulting candidate ($W_1 = 0.226$) identifies `thal` as a branching variable and closely tracks the real conditional distribution of `num` in both branches (Fig. 3, center and right).

## 6 Conclusions

We presented ALPS, a PP synthesis system that couples LLM-driven structural search with gradient-based parametric optimization, replacing the blind variation operators of prior evolutionary and MCMC-based approaches with proposals conditioned on optimization feedback. Across nine benchmarks, ALPS achieves the best likelihood fit on average on all of them and remains competitive in distributional distance to the ground truth, while also showing promising signs of recovering the correct dependency structure between variables, as illustrated by our qualitative analysis. Our ablations confirm that LLM capability, iterative mutation, and gradient refinement each contribute meaningfully to performance. We conducted a real-world case study, confirming that ALPS can recover meaningful structure from real, noisy data and that prior knowledge, injected as natural-language hypotheses, can drive the synthesis. Extending to higher-dimensional, more realistic datasets remains an important direction for future work, which may benefit from larger and more capable LLM backends to handle the increased structural complexity of the search space. A further direction is applying ALPS to the synthesis of structural causal models, formulated as PPs, by extending our pipeline to conditioned LLM-guided structural search on interventional or counterfactual queries in addition to observational fit.

## AI USE STATEMENT

In this work, we used generative AI tools for editing software code and improving the readability of the research paper. We have not used generative AI tools for conducting experiments, analyzing results, or generating scientific claims, and the use of generative AI for data collection and annotation is not applicable to this work. Additionally, we used generative AI tools to assist in identifying relevant literature. We have reviewed all AI-assisted work: LLM-generated or LLM-edited code was verified and tested for correctness by the authors, literature suggestions were manually checked for relevance and accuracy, and all AI-assisted edits to the paper text were reviewed and approved by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

## ETHICS STATEMENT

This work presents a method for automated probabilistic program synthesis from observational data. It does not involve human subjects or data collection of any kind. Most benchmarks used are synthetic datasets generated from known probabilistic programs; the one real-world dataset used (the UCI Heart Disease dataset (Janosi & Detrano, 1989)) is public, de-identified, and anonymized, containing no personally identifiable information. We are not aware of any potential harms, conflicts of interest, fairness concerns, or legal compliance issues arising from this research.

## REPRODUCIBILITY STATEMENT

To support reproducibility, we provide a complete description of the ALPS pipeline in Section 3 and Algorithm 1, including all hyperparameters in Appendix A.2 and the full LLM prompts used for generation and mutation in Appendix A.1. All baselines are described in detail in Appendix C, with hyperparameters and implementation choices reported for each. The benchmarks used in our experiments are fully described in Appendix A and Table 3, and are drawn from publicly available sources. The source code of ALPS is submitted as anonymous supplementary material.

## REFERENCES


Dhruv Agarwal, Bodhisattwa Prasad Majumder, Reece Adamson, Megha Chakravorty, Satvika Reddy Gavireddy, Aditya Parashar, Harshit Surana, Bhavana Dalvi Mishra, Andrew McCallum, Ashish Sabharwal, and Peter Clark. Autodiscovery: Open-ended scientific discovery via bayesian surprise. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Sarah E. Chasins and Phitchaya Mangpo Phothilimthana. Data-driven synthesis of full probabilistic programs. In Rupak Majumdar and Viktor Kuncak (eds.), *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pp. 279–304. Springer, 2017. doi: 10.1007/978-3-319-63387-9_14. URL https://doi.org/10.1007/978-3-319-63387-9_14.

CodeGemma Team. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Julie Delon and Agnès Desolneux. A Wasserstein-type distance in the space of Gaussian Mixture Models. *SIAM Journal on Imaging Sciences*, 13(2):936–970, 2020. doi: 10.1137/19M1301047. URL https://hal.science/hal-02178204.

Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

Justin Domke. Large language bayes. *arXiv preprint arXiv:2504.14025*, 2025.

Romina Doz, Francesca Randone, Eric Medvet, and Luca Bortolussi. Evolutionary synthesis of probabilistic programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2025. doi: 10.1145/3712256.3726388.

Romina Doz, Francesca Randone, Luca Bortolussi, and Eric Medvet. Inferring structural causal models from data with grammatical evolution. In Giovanni Iacca, Giorgia Nadizar, Anil Yaman, Doina Bucur, Antonio Della Cioppa, Ting Hu, Eric Medvet, and Sarah L. Thomson (eds.), *Parallel Problem Solving from Nature – PPSN XIX*, pp. 119–134, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-36229-2.

Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1194–1201, 2017.

Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

Simos Gerasimou, Giordano Tamburrelli, and Radu Calinescu. Search-based synthesis of probabilistic models for quality-of-service software engineering. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 319–330, 2015.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 167–181. ACM, 2014. doi: 10.1145/2593882.2593900.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Steinbrunn William Pfisterer Matthias Janosi, Andras and Robert Detrano. Heart Disease. UCI Machine Learning Repository, 1989. DOI: https://doi.org/10.24432/C52P4X.

Madhav Kanda, Shubham Ugare, and Sasa Misailovic. Refinestat: Efficient exploration for probabilistic program synthesis. *arXiv preprint arXiv:2509.01082*, 2025.

Michael Y. Li, Emily B. Fox, and Noah D. Goodman. Automated statistical model discovery with language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pp. 50–60, 1947.

Osvaldo A. Martin, Oriol Abril-Pla, Jordan Deklerk, Seth D. Axen, Colin Carroll, Ari Hartikainen, and Aki Vehtari. Arviz: a modular and flexible library for exploratory analysis of bayesian models. *Journal of Open Source Software*, 11(119):9889, 2026. doi: 10.21105/joss.09889. URL https://doi.org/10.21105/joss.09889.

Jeffrey Morgan and Michael Chiang. Ollama, September 2026. URL https://github.com/ollama/ollama.

Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 208–217, 2015.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro, 2019. URL https://arxiv.org/abs/1912.11554.

Aaditya Ramdas, Nicolas Garcia, and Marco Cuturi. On wasserstein two sample testing and related families of nonparametric tests, 2015. URL https://arxiv.org/abs/1509.02237.

Francesca Randone, Luca Bortolussi, and Mirco Tribastone. Inference of probabilistic programs with moment-matching gaussian mixtures. *arXiv preprint arXiv:2311.08235*, 2023.

Francesca Randone, Romina Doz, Mirco Tribastone, and Luca Bortolussi. Degas: Gradient-based optimization of probabilistic programs without sampling. *arXiv preprint arXiv:2601.15167*, 2026.

Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *arXiv preprint arXiv:2405.20132*, 2024.

Adam Viktorin, Tomas Kadavy, Jozef Kovac, Michal Pluhacek, and Roman Senkerik. EASE-ing into global optimization with LLMs. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '25 Companion, New York, NY, USA, 2025. Association for Computing Machinery. doi: 10.1145/3712255.3735096.

Stefan Wahl, Raphaela Schenk, Ali Farnoud, Jakob H. Macke, and Daniel Gedon. A probabilistic framework for llm-based model discovery. *arXiv preprint arXiv:2602.18266*, 2026.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020. URL https://arxiv.org/abs/1910.03771.

Lionel Wong, Katherine M. Collins, Lance Ying, Cedegao E. Zhang, Adrian Weller, Tobias Gerstenberg, Timothy O'Donnell, Alexander K. Lew, Jacob D. Andreas, Joshua B. Tenenbaum, and Tyler Brooke-Wilson. Modeling open-world cognition as on-demand synthesis of probabilistic models. *arXiv preprint arXiv:2507.12547*, 2025.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

# A Implementation Details

## A.1 LLM Prompts

This section reports the templates used to query the LLM Module at both stages of the pipeline (LLM Generation and LLM Mutation). Both prompts request a response in a fixed JSON schema.

**DeGAS grammar reference.** The following specification is embedded verbatim in the system prompt for both LLM Generation and LLM Mutation. It restates the DeGAS grammar in a compact form, together with a short list of common mistakes we observed the LLM to make in preliminary runs, which we found reduced the rate of invalid candidates reaching DeGAS Evaluation. It also gives explicit guidance for exactly-discrete/binary variables: DeGAS's own representation of a Bernoulli-like variable is a zero-variance point mass, which is unreachable by an unconstrained continuous `sigma_i > 0` constraint and, we found, was otherwise a target for gradient descent to drive `sigma_i` arbitrarily close to zero to exploit the unbounded density of a near-degenerate Gaussian component – inflating likelihood on a specific data point rather than genuinely fitting the distribution.

```
DEGAS Language Reference:
STRUCTURE
- Instructions end with ;
- Two instruction types: assignment (var = expr;) and
  conditional (if condition { program } else { program } end if;)
- Only dataset variables are usable (see var_names in the stats)

NUMBERS
- Decimals with at most 2 decimal places, range [-100.00, 100.00]
- Weights and standard deviations must be > 0

DISTRIBUTIONS
- gm([pi_1,...,pi_n], [mu_1,...,mu_n], [sigma_1,...,sigma_n])
  - All three lists same length, n >= 1
  - pi_i > 0 and sum(pi) = 1.00
  - sigma_i >= 0
  - mu_i any number in [-100, 100]
- uniform([start, end], 2)
  - start < end, both in [-100.00, 100.00]
  - the trailing 2 is a fixed literal, never change it

DISCRETE / BINARY VARIABLES
- If a variable only takes a small number of exact values in the
  data (e.g. a 0/1 indicator), represent each value as its own
  gm(...) component with sigma_i = 0.00 -- an exact point mass at
  mu_i, not an approximation.
  Example: a variable that is 0 about 70% of the time and 1 about
  30% of the time -> gm([0.70, 0.30], [0.00, 1.00], [0.00, 0.00])
- Do NOT use a small positive sigma (like 0.01) to approximate a
  discrete value -- only sigma_i = 0.00 is treated as an exact
  point mass; any nonzero sigma is scored as a continuous density,
  which is undefined behavior here.
- IMPORTANT: pi_i and mu_i of a sigma_i = 0.00 component are never
  refined afterward -- there is no gradient for a point mass in
  DeGAS. Set them as accurately as you can directly from the data
  summary (e.g. pi_i = the empirical fraction of observations at
  that exact value); they will not be corrected later.

ASSIGNMENTS (key constraint)
- var = expr;
- At most ONE multiplication (*) per line, no division allowed
- If a product has a number and a variable, the number must
  come first (3.00*b, not b*3.00)
- Legal forms: atom | number*var | number*dist | var*var |
  atom+atom | atom-atom | number*var + atom | number*var - atom |
```

```
  atom + number*var | atom - number*var | number*dist + atom |
  atom + number*dist
  (atom = var | number | distribution)
- Two products need a temp variable:
  INVALID: x = 2.00*y + 3.00*z;
  VALID:   var1 = 2.00*var2;  var1 = var1 + 3.00*var3;

CONDITIONALS
- if condition { program } else { program } end if;
- condition forms: var == number | var != number | lexpr < number |
  lexpr <= number | lexpr >= number | lexpr > number
- lexpr = var

COMMON MISTAKES TO AVOID
- Normal(0,1) syntax is invalid -> use gm([1.00],[0.00],[1.00])
- Weights must sum exactly to 1.00 at 2 decimal places
- No division (b/c is invalid)
- Number must come first in products (var1*3.00 -> 3.00*var1)
- No more than one multiplication per line -> split with a temp variable
- uniform([0,1]) is invalid -> must be uniform([0.00, 1.00], 2)
```

Immediately after this reference, the system prompt closes with a short rules recap that restates the constraints most often violated in practice and fixes the required output format:

```
RULES
- Numbers: exactly 2 decimal places, range [-100.00, 100.00]
- gm weights must sum to exactly 1.00
- At most one * per assignment line
- Coefficient always before variable in products
- No division
- Do NOT put underscores or other characters in variable names, use
  the names of the dataset variables exactly as they appear in the stats
- Reply ONLY with a valid JSON object -- no prose, no markdown fences
```

**Initialization prompt (LLM Generation).** The user-turn message is built from the observed dataset's variable names and per-variable summary statistics, together with a fixed set of structural templates, and requests exactly $K$ structurally diverse candidates, one per template:

```
Data summary:
var_names:  [list of dataset variable names]
n:  [sample size]
mean:  [per-variable mean]
std:  [per-variable standard deviation]
skewness:  [per-variable skewness]
kurtosis:  [per-variable kurtosis]
min:  [per-variable minimum]
max:  [per-variable maximum]
Generate exactly K DeGAS programs that are STRUCTURALLY
DIVERSE and include all the variables of the dataset
(var_names in the stats).
Aim for one program per structure type below:
1.  unimodal --- single gm component, match the data mean
and std
2.  mixture2 --- two-component gm, one per apparent
subpopulation
3.  mixture3 --- three-component gm for skewed or
multi-modal data
4.  conditional --- if/else branching
5.  hierarchical --- product of two distributions (use temp
variable)
For each program:  (1) write a one-sentence hypothesis
about the data-generating process; (2) choose distribution
families matching the data range and shape; (3) write valid
```

```
DeGAS code --- remember at most one * per line.

Return ONE JSON object with the exact shape below (no prose,
no markdown):
{"programs": [{"id": int, "hypothesis": str, "structure":
str, "program": str}]}
```

When testing dependency hypotheses on specific variables, one or more of the five canonical slots above are replaced in place with a targeted hint, e.g. slot 2 becomes "`hypothesis --- assume` $y$ `depends primarily on` $x$`; encode this dependency structurally (e.g. via if/else or a shared temp variable)`", leaving the remaining canonical slots and $K$ unchanged.

**Mutation prompt (LLM Mutation).** The prompt opens by framing the current step as one iteration of the synthesis-and-refinement loop:

```
You are helping improve a set of probabilistic programs
that model a dataset.  This is iteration [iteration] of a
synthesis-and-refinement loop, in which at each step you
propose [n_mutations] program and I optimize some of their
parameters with gradient descent and report back the final
loss and parameter values.
```

The pool of surviving candidates is then serialized into a per-candidate diagnostic block reporting the loss before and after gradient refinement, the initial-to-optimized value of every continuous parameter, and a deterministic heuristic hint derived from these deltas:

```
### Program [rank]  [id=[id]]
hypothesis:  [one-sentence hypothesis]
loss:  [initial] → [final]  (Δ [delta])
parameters (initial → after gradient):
[param name]:  [initial value] → [optimized value]  (Δ
[delta])
suggested mutation focus:  [heuristic hint, e.g. ''large
parameter movement --
consider whether this signals a missing component'']
program:
[DeGAS program text]
```

This is followed by the task instructions, including an explicit id mapping that assigns each requested mutation's output id deterministically (avoiding leaving id arithmetic to the LLM):

```
For EACH program in the pool below, propose EXACTLY ONE
mutated program that is likely to achieve a lower loss
than that program.  You must produce [n_mutations] mutated
programs total, one per parent, using the id mapping below:

- parent_id=[id] (Program [rank] above) -> use id=[assigned
id]

Focus on structural mutations --- the gradient optimizer
will handle numeric fine-tuning after you.  Avoid simply
changing numbers.

Useful structural mutations to consider (use the ''suggested
mutation focus'' for each program to pick the most
relevant ones):  split a component; merge components; add
a conditional; remove a conditional; change dependency
structure (introduce or remove intermediate variables to
capture correlations between variables); re-initialise a
poor program with a different hypothesis.

Across the returned mutated programs, use at least 2
distinct structure tags.  Do not return programs that are
all minor variations of the same structural idea.

Other rules:  do not put underscores or other characters in
```

```
variable names; you may reuse and modify the best-performing
programs as starting points, or generate entirely new
structures; do not copy a program unchanged; initialise
numeric parameters to reasonable values --- the gradient
will refine them, but a good starting point helps.

Reply ONLY with a valid JSON array of objects, each with
fields:  id, parent_id, hypothesis, structure, program.
```

### A.2 HYPERPARAMETER SELECTION

**Population and mutation size.** The $K = 5$ initial candidates correspond one-to-one with the five structural templates used to seed LLM Generation (unimodal, mixture2, mixture3, conditional, hierarchical), so the initial population is guaranteed to span these five qualitatively different hypotheses rather than being an arbitrary sample size. The 5 mutations requested per generation likewise match the top-$M = 5$ pool size retained by Pool Update, so that every surviving candidate receives exactly one mutated offspring per iteration.

**Gradient refinement schedule.** The number of Adam steps increases across the two points at which a candidate is refined: $i = 100$ after mutation and a further $j = 200$ for candidates retained in the top-$M$ pool, reflecting increasing confidence in the candidate. Only candidates that survive Pool Update and persist across iterations receive the larger, additional refinement budget. The learning rate is set to $0.01$ across all the refinement stages.

**Stopping criterion.** The threshold $\epsilon = 0.01$ was chosen to treat a loss improvement below it as noise rather than genuine progress throughout ALPS. The patience $p = 3$ follows standard practice for early-stopping windows in iterative optimization: short enough to react promptly once progress has genuinely stalled, but long enough to tolerate the run-to-run fluctuation inherent to LLM-proposed mutations. The cap $T = 10$ was set empirically looking at the average loss during iterations, observing that further mutations typically stopped yielding meaningful improvement beyond this point.

**Sampling temperature.** LLM Generation uses temperature $0.2$ and LLM Mutation uses temperature $0.4$. The asymmetry follows from what each stage needs to get right: LLM Generation must reliably produce $K$ candidates that are both grammar-valid and structurally diverse across fixed templates, so a lower temperature favors closer adherence to those instructions over the grammar-parsing failures a higher temperature tends to introduce. LLM Mutation instead starts from an already-valid parent program and is explicitly asked for structural exploration (splitting or merging components, adding or removing conditionals, changing dependency structure), which benefits from more sampling diversity; a failed or unproductive mutation is also lower-cost at this stage, since the parent and the rest of the candidate pool survive regardless. Both values were set from this reasoning and preliminary runs rather than a systematic temperature sweep.

**LLM backend and model scale.** ALPS's default configuration uses `gpt-oss:120b` (116.8B parameters), served via a self-hosted Ollama instance. This is substantially larger than the 7–8B model class used for the RefineStat baselines, whose SynCode grammar-constrained decoding enforces syntactic validity at generation time and so does not require a large model to produce well-formed programs. ALPS instead relies on the model following DeGAS's grammar from prompt instructions alone, which our backend-scale ablation (Sec. 5) found to depend heavily on both model scale and family: two of eight tested backends, spanning four model families and 7–120B parameters, never produced a single candidate that survived DeGAS compilation across any of 5 seeds, and two more succeeded only intermittently, while `gpt-oss:120b` achieved both the lowest per-program error rate ($4.6\% \pm 6.3\%$) and the best NLL ($4.02 \pm 0.06$) and $W_2^2$ distance to the ground-truth program ($8.31 \pm 9.03$) among the backends that succeeded, on the `if` running example.

**Compute infrastructure.** LLM-based generation is served remotely: an NVIDIA GB10 (Grace Blackwell, unified CPU/GPU memory, ≈128 GB, Ubuntu 24.04) machine hosts an Ollama server for ALPS's LLM calls and runs RefineStat-DeGAS's SynCode-constrained Qwen2.5-Coder-7B-Instruct generation together with both of its inference backends (gradient and NumPyro-based

MCMC). All non-LLM computation for ALPS (DeGAS gradient optimization) and the entirety of grammar-MCMC (no LLM involved) run locally on a 16-thread AMD Ryzen 7 PRO 5850U with 14 GB RAM. Seeds and benchmarks are run sequentially within each method, both to avoid contention on the shared GPU server and because refinement attempts (bounded by $R_{max}$) are themselves inherently sequential.

### A.3 Structured Output and Failure Recovery

Every call to the LLM backend requests output constrained to a fixed JSON schema via Ollama's structured-output decoding (the `format` field of the request), rather than free-form text. The schema requires a single top-level object,

```
{"programs": [{"id": int, "hypothesis": str,
"structure": str, "program": str}]},
```

so that every candidate returned by the model carries an explicit hypothesis and structure tag alongside its DeGAS code, regardless of whether the call is for initial candidate generation or for mutation.

**Response extraction.** Since different backends and chat templates expose generated text under different keys, the response is extracted defensively: the pipeline checks, in order, a top-level `response` field, a chat-style `message.content` field, and – for reasoning models that can emit their answer under a `message.thinking` field when `content` is empty – that field as a fallback, before finally checking a bare `content` field.

**Parsing.** The extracted text is first parsed as JSON directly. If that fails (e.g. the model wrapped its answer in prose or a markdown fence despite the schema constraint), the pipeline falls back to a regex search for the first `{...}` or `[...]` span in the text and parses that instead, recovering well-formed JSON embedded in otherwise unstructured output.

**Corrective retry.** If parsing still fails, the pipeline issues a corrective retry: a new call to the same model, at temperature 0 (rather than the sampling temperature used for the original call, to reduce the chance of a second malformed response) and with the user message replaced by an explicit restatement of the required JSON shape together with the original task. This is attempted up to `max_retries` times (one, in all experiments reported here).

**Salvage step.** If the corrective retry is exhausted and parsing still fails, a final salvage step is attempted: rather than repeating the original task, the model is instead asked, at temperature 0 and under the same schema constraint, to convert its own last raw (unparseable) response into valid JSON matching the required shape, explicitly permitted to keep only the entries it can recover and to assign missing ids sequentially. This can rescue partially-formed output – e.g. a response containing several valid program entries alongside one truncated or malformed one – that a strict reparse of the original text would discard entirely.

**Discarding.** Only if the salvage step's own output also fails to parse is the call treated as a total failure: it returns an empty program list together with a recorded parse error.

**Transport-level retries.** This failure-recovery chain is distinct from, and sits on top of, a separate transport-level retry mechanism: connection errors and read timeouts against the Ollama server are retried up to `max_http_retries` times with a linearly increasing backoff, independent of whether the eventual response parses correctly. A transport failure that exhausts its own retry budget is reported immediately and does not trigger the JSON corrective-retry or salvage steps described above, since no response text was received to recover from.

## B Benchmarks

Table 3 lists the nine target programs used throughout our experiments. Each defines a ground-truth probabilistic program in DeGAS, from which we forward-sample a synthetic dataset that is

the only input every synthesis method receives – none of the methods have access to the program itself, only to samples drawn from it. The same compiled target program also serves as the ground-truth reference for the Gaussian-Wasserstein ($W_2^2$) distance reported in our results, measuring how closely a synthesized program's joint distribution matches the true generative process in parameter space, independent of the specific dataset drawn from it.

The suite was designed to span the structural dimensions DeGAS programs can vary along. It ranges from $m = 2$ to $m = 4$ variables; from purely continuous programs built entirely from Gaussian mixtures (*IfModel*, *MoG1*, *MultipleBranches*) to purely discrete ones built from Bernoulli and categorical point masses (*Eyecolor*, *Hurricane*, *Csi*), with three mixed-type programs in between (*EasyTugWar*, *BiasedTugWar*, *MixedCondition*); and across dependency structures of increasing complexity, from a single conditional branch (*IfModel*) through nested branching on both a discrete and a continuous variable (*MixedCondition*) and cascading branches (*MultipleBranches*), to hierarchical parameter-sharing (*MoG1*) and a four-variable discrete Bayesian network (*Csi*). This coverage lets us attribute a method's success or failure on a given benchmark to a specific structural property (e.g. discrete variables, nested conditionals, or hierarchical dependencies).

Table 3: The target programs used for the experiments. G., B., and C. stand for Gaussian, Bernoulli, and categorical distributions; $m$ is the number of variables; Type indicates whether the program's variables are all continuous, all discrete, or a mixture of both.

| Name | Description | $m$ | G. | B. | C. | Type |
|---|---|---|---|---|---|---|
| *If* | Running example described in section 2.2 | 2 | ✓ | | | Continuous |
| *MoG* | Defines three normally distributed variables, in which two are used as parameters for the third. | 3 | ✓ | | | Continuous |
| *EasyTugWar* | Assigns the victory to the player with higher skills, given normally distributed skills. | 3 | ✓ | ✓ | | Mixed |
| *BiasedTugWar* | Variation of EasyTugWar. | 3 | ✓ | ✓ | | Mixed |
| *MixedCondition* | Nested if branch depending on a discrete and a continuous variable. | 3 | ✓ | ✓ | | Mixed |
| *MultipleBranches* | Models the length of questions after lecture, given the difficulty of the content. | 2 | ✓ | | | Continuous |
| *Eyecolor* | Models the distributions of eye color, hair color and heir length. | 3 | | | ✓ | Discrete |
| *Hurricane* | Models the level of damage depending on the preparation level of a city. | 2 | | | ✓ | Discrete |
| *Csi* | A Boolean Bayesian network. | 4 | | | ✓ | Discrete |

# C Baselines

## C.1 Grammar-MCMC

The grammar-MCMC baseline (`MCMC` in the main results) performs blind Metropolis-Hastings search directly over the space of DeGAS programs, with no LLM involvement at any stage. Its hyperparameters, and the role each plays, are as follows.

**Search length:** $n_steps = 10^4$**.** The number of proposal/accept-reject iterations. We set this high to give the baseline a large computational budget and make it as competitive as possible, though the method was not able to obtain competitive results with the other baselines.

**Initialization:** $p_conditional_init = 0.3$**,** $max_init_tries = 50$**.** The initial program is built by assigning each variable, in turn, either an unconditional distribution or a conditional (`if/else`) statement over an earlier variable, chosen independently per variable with probability $0.3$ of being conditional. If no fittable initial program is found within $50$ such attempts, the run is abandoned as a failure for that seed.

**Proposal mechanism.** At each step, one of three structural mutations is chosen uniformly at random: *resample distribution* (redraw a variable's distribution parameters from the prior), *toggle conditional* (convert an unconditional statement into a conditional one by introducing a fresh random condition and second branch, or collapse an existing conditional back to unconditional), or *resample condition* (redraw only the condition variable/operator/threshold of an existing conditional statement, chosen uniformly among the program's conditional statements). If the chosen mutation is a no-op in the current state (e.g. toggling a conditional on a variable with no earlier variable to condition on, or resampling a condition when none exists yet), a different mutation type is retried up to $10$ times before falling back to *resample distribution*, which is always applicable. This is the only mechanism by which structure can change; there is no LLM proposing or biasing these moves in any way.

**Parameter fitting:** $n_opt_steps = 50$**,** $lr = 0.01$**.** Once a structural proposal is fixed, its continuous parameters are fit by the same gradient-based optimizer used in ALPS, for $50$ steps at learning rate $0.01$, before the proposal's fitted negative log-likelihood is used to decide acceptance.

**Acceptance:** $temperature = 1.0$**.** The fitted proposal is accepted following the standard Metropolis criterion, based on *training* loss: unconditionally if it improves (or does not worsen) the training loss, and otherwise with probability $\exp(-\Delta/temperature)$, where $\Delta$ is the increase in training loss over the current state. At $temperature = 1.0$, this is an unscaled Metropolis acceptance rule with no additional annealing. Independently of acceptance, the best candidate reported at the end of a run is the one with the lowest NLL (computed on the training data) seen at any point in the chain.

### C.2 REFINESTAT FOR DEGAS

We adapt RefineStat's semantically-constrained generation and D‖P‖L diagnostic-aware refinement loop (Kanda et al., 2025) to DeGAS. The original codebase could not be reused directly: its checkers validate code via Python `ast`/`importlib` reflection over live PyMC classes, which does not apply to DeGAS's non-Python grammar and its inference relies on `pm.sample()`, and DeGAS has no native MCMC engine. We therefore reimplemented part of the algorithm, including the refinement loop, the six validity-predicate categories, and default diagnostic thresholds – matching RefineStat's own reported configuration as closely as DeGAS's execution model allows, as detailed below.

**Generation.** Candidates are produced by SynCode grammar-constrained decoding (`grammar_strict` mode) with Qwen2.5-Coder-7B-Instruct, matching RefineStat's class of 7–8B open-weight SLMs, over the same restricted DeGAS grammar our method's prompt enforces. The generation unit is one top-level statement (an assignment or a full `if/else` block); a failed check backtracks one unit rather than restarting. The six validity predicates are reimplemented as a parse-tree walk rather than via Python reflection.

**Two inference backends.** DeGAS's native inference is gradient-based point estimation, not posterior sampling, so we report two variants, isolating RefineStat's generation/refinement strategy (shared by both) from its posterior-sampling inference (present only in the second).

**RS-Grad** fits the parameters of each candidate PP by gradient descent (Adam), with the identical parameter-fitting procedure used by ALPS. It is therefore fast to run, but it does not reproduce RefineStat's actual Bayesian posterior-sampling inference: there is no posterior to diagnose, so its

five acceptance checks (R-hat, bulk/tail ESS, divergence rate, held-out fit) are instead computed over independent forward (ancestral) simulations of the fitted point estimate and BFMI and Pareto-$k$/LOO are dropped entirely, as both are specific to an actual HMC posterior that a point estimate does not have. This backend isolates the effect of RefineStat's generation/refinement strategy alone, with the inference mechanism held fixed and identical to our method's.

**RS-MCMC** instead translates each candidate into a NumPyro (Phan et al., 2019) model, in order to reproduce RefineStat's own seven-check reliability score in full. GMs are represented as `MixtureSameFamily`. Since this class involves a non-differentiable step for choosing which component to draw from, it cannot be exposed to NUTS as a free latent variable. To handle observed variables assigned inside `if`/`else` blocks, we therefore compute the log-likelihood of the observed value under each branch separately, combine the two using the branch condition as a mask, and add the result directly to the model's log-probability via `numpyro.factor`. Free parameters receive weakly-informative, data-agnostic priors, $\mu \sim \mathcal{N}(0, 50)$ and $\sigma \sim \text{HalfNormal}(25)$, spanning `DEGAS_GRAMMAR`'s declared $[-100, 100]$ range without conditioning on the dataset. Chains run natively in parallel via NumPyro's vectorized multi-chain execution. Diagnostics are computed through ArviZ (Martin et al., 2026).

**Hyperparameters.**

- Refinement loop: $R_{max} = 100$, $\alpha = 2$ (likelihood resamples), $\beta = 4$ (target valid candidates) – RefineStat's own reported values, used identically for *both* backends to give them a matched search budget.
- Acceptance cutoff: $K = 4/5$ passing checks for the gradient backend (its own reduced, forward-simulation-based diagnostic set); $K = 5/7$ for the MCMC backend, i.e. $\zeta = 5$ over RefineStat's own seven checks reproduced exactly.
- Gradient backend: 100 Adam steps, learning rate $0.01$.
- MCMC backend: 4 chains, 1000 warmup + 1000 sampling steps, matching RefineStat's own `pm.sample(1000, tune=1000, chains=4, ...)` call exactly. A candidate is accepted once $\geq 5$ of the following 7 checks pass: $\hat{R} < 1.05$, ESS bulk $\geq 400$, ESS tail $\geq 100$, divergence rate $\leq 5\%$, BFMI $> 0.3$, Pareto-$k > 0.7$ in $\leq 20\%$ of observations, and successful LOO computation – all thresholds identical to RefineStat's own defaults.

### C.3 Evolutionary Baseline

The Evolutionary baseline (Doz et al., 2025) uses grammatical evolution (GE) as implemented in PonyGE2 (Fenton et al., 2017), with the hyperparameters reported by the original authors. A population of $n_{\text{pop}} = 200$ individuals is initialized as variable-length integer genotypes $\boldsymbol{g} \in \{0, \ldots, n_{\max}\}^l$, with $n_{\max} = 500$ and $l = 200$, and evolved for $n_{\text{gen}} = 500$ generations. At each generation, $n_{\text{pop}}$ offspring are produced by repeating $n_{\text{pop}}$ times: with probability $p_{\text{xover}} = 0.75$, two parents are selected via tournament selection (size 3) and combined via two-point crossover; otherwise, one parent is selected via tournament selection and gene-flip mutation is applied, flipping each element of $\boldsymbol{g}$ to a different value in $\{0, \ldots, n_{\max}\}$ with probability $1/|\boldsymbol{g}|$. The top $n_{\text{elite}} = 10$ parents by fitness are retained and merged with the best $n_{\text{pop}} - n_{\text{elite}}$ offspring to form the next generation's population.

Fitness is computed as the average log-likelihood over a mini-batch of $n' = 100$ points sampled from the fixed 1000-point training set. A program is assigned fitness $-\infty$ if it fails to declare all dataset variables, if evaluated data points fall outside the program's support, or if fitness evaluation exceeds a $t_{\max} = 20\,\text{s}$ timeout.

This gives the Evolutionary baseline $n_{\text{pop}} \times n_{\text{gen}} = 200 \times 500 = 100{,}000$ candidate program evaluations per run.

Table 4: Mean rank (1 = best, 5 = worst) of each method across the 9 benchmarks in Table 1, with the Friedman test statistic and $p$-value for the null hypothesis that all methods are equivalent.

| Metric | ALPS | RS-MCMC | RS-Grad | MCMC | Evolut. | $\chi^2$ (df=4) | $p$ |
|---|---|---|---|---|---|---|---|
| NLL | **1.00** | 3.33 | 3.00 | 4.22 | 3.44 | 20.89 | 0.0003 |
| $W_2^2$ | **1.67** | 3.00 | 2.67 | 4.33 | 3.33 | 13.60 | 0.0087 |

Table 5: Wilcoxon signed-rank test, ALPS vs. each baseline, paired by benchmark, on the mean values from Table 1. $n$ is the number of benchmarks where both methods converged. Bold: $p < 0.05$.

| | NLL | | $W_2^2$ | |
|---|---|---|---|---|
| vs. | $n$ | $p$ | $n$ | $p$ |
| RS-MCMC | 7 | **0.0156** | 7 | 0.2969 |
| RS-Grad | 8 | **0.0078** | 8 | 0.0781 |
| MCMC | 9 | **0.0039** | 9 | **0.0039** |
| Evolutionary | 8 | **0.0078** | 8 | **0.0391** |

# D Additional Results

## D.1 Statistical significance of the cross-method comparison

Table 1 reports mean $\pm$ std over 5 independent runs per method per benchmark. To assess whether ALPS's apparent advantage reflects a systematic effect rather than noise from these small samples, we apply three complementary non-parametric tests: a Friedman test (an omnibus check across all 5 methods), a Wilcoxon signed-rank test (ALPS vs. each baseline, paired across benchmarks by their mean), and an exact Mann-Whitney U test (ALPS vs. each baseline, within each benchmark, on the raw per-run values) – the first two following the methodology for comparing multiple methods across multiple datasets of Demšar (2006), the third added because the first two treat each benchmark's mean as a single observation and are blind to the spread of the underlying runs. We avoid parametric tests (e.g. a paired $t$-test) throughout since several cells in Table 1 exhibit strong right-skew and outlier-dominated variance (e.g. Evolutionary's $W_2^2 = 26341 \pm 58808$ on `BiasedTugWar`, where the std exceeds the mean), violating the normality assumption those tests require.

**Friedman test.** The Friedman test (Friedman, 1937) is a non-parametric analogue of a repeated-measures ANOVA: for each benchmark ("block"), the 5 methods are ranked 1 (best) to 5 (worst) on the given metric, and the test asks whether the average rank differs across methods by more than would be expected by chance, under the null hypothesis that all methods are exchangeable. A method that did not converge on a given benchmark (*n.c.* in Table 1) is assigned the worst rank (5) on that benchmark. Table 4 reports the resulting mean ranks and the Friedman test statistic.

Both tests reject the null hypothesis of equivalent methods at $p < 0.01$. ALPS's mean rank of exactly $1.00$ on NLL means it is ranked best on *every one* of the 9 benchmarks; on $W_2^2$ its mean rank of $1.67$ reflects that it is best on most, but not all, benchmarks (`MoG`, `MixedCond.`, and `Hurricane`/`CSI` are won by RS-MCMC, RS-Grad, and Evolutionary respectively).

**Wilcoxon signed-rank test.** The Friedman test only establishes that *some* difference exists among the 5 methods; to test ALPS specifically against each baseline, we use the Wilcoxon signed-rank test (Wilcoxon, 1945), a non-parametric paired test on the per-benchmark differences between ALPS and a given baseline (paired by benchmark, since both methods are evaluated on the same 9 data-generating processes). Benchmarks where the baseline did not converge are excluded from that specific pairwise test (reducing $n$ below 9), since there is no value to pair against. Table 5 reports the resulting $p$-values.

On NLL, ALPS is significantly better than every baseline ($p < 0.05$ throughout, and $p < 0.01$ against three of the four). On $W_2^2$, the comparison is more nuanced: ALPS is significantly better than MCMC (`grammar_mcmc`) and Evolutionary, but the difference against RS-MCMC and RS-

Table 6: Exact Mann-Whitney U test $p$-values, ALPS vs. each baseline, within each benchmark, on raw NLL. Bold: $p < 0.05$.

| Benchmark | RS-MCMC | RS-Grad | MCMC | Evolutionary |
|---|---|---|---|---|
| If | **0.0079** | **0.0079** | **0.0079** | **0.0079** |
| MoG | **0.0159** | **0.0317** | **0.0079** | **0.0079** |
| CSI | **0.0079** | n.c. | **0.0079** | 1.0000 |
| EasyTugWar | n.c. | **0.0079** | **0.0079** | **0.0079** |
| BiasedTugWar | n.c. | **0.0079** | **0.0079** | 0.0952 |
| MixedCond. | **0.0079** | **0.0159** | **0.0079** | 0.3095 |
| MultiBranch | **0.0079** | **0.0079** | **0.0079** | **0.0079** |
| EyeColor | **0.0079** | **0.0079** | **0.0079** | n.c. |
| Hurricane | **0.0079** | **0.0079** | **0.0079** | 0.4206 |

Table 7: Exact Mann-Whitney U test $p$-values, ALPS vs. each baseline, within each benchmark, on raw $W_2^2$. Bold: $p < 0.05$.

| Benchmark | RS-MCMC | RS-Grad | MCMC | Evolutionary |
|---|---|---|---|---|
| If | 0.6905 | **0.0079** | **0.0079** | **0.0079** |
| MoG | **0.0079** | 0.3095 | 0.0556 | 0.5476 |
| CSI | 0.1508 | n.c. | **0.0079** | **0.0079** |
| EasyTugWar | n.c. | 0.8413 | **0.0079** | 0.5476 |
| BiasedTugWar | n.c. | 0.5476 | 0.1508 | 0.3095 |
| MixedCond. | 0.8413 | 0.6905 | **0.0079** | **0.0159** |
| MultiBranch | **0.0317** | **0.0079** | **0.0079** | **0.0079** |
| EyeColor | 1.0000 | 0.1508 | **0.0079** | n.c. |
| Hurricane | 0.0556 | **0.0079** | **0.0079** | 0.5476 |

Grad does not reach significance at this sample size – consistent with Table 1, where RS-MCMC wins $W_2^2$ outright on `MoG` and is competitive on several other benchmarks. We note that with only 7–9 paired observations, this test on benchmark means is underpowered and blind to within-benchmark variance; the within-benchmark Mann-Whitney tests below directly address both limitations using the raw per-run values.

**Within-benchmark Mann-Whitney U tests.** As a variance-aware complement, we also run an exact Mann-Whitney U test (Mann & Whitney, 1947) within each benchmark, comparing ALPS's 5 runs directly against each baseline's runs on the raw NLL and $W_2^2$ values. Benchmarks where a baseline did not converge are excluded for that baseline, as above. Tables 6 and 7 report the resulting $p$-values.

Against RS-MCMC, RS-Grad, and MCMC, ALPS's NLL advantage is significant on every benchmark where a comparison is possible (24 of 24). Against Evolutionary, however, the advantage is significant on only 4 of 8 testable benchmarks (`if`, `mog`, `easytugwar`, `multibranch`) – and fails to reach significance on precisely the benchmarks with discrete-valued variables (`csi`, `biasedtugwar`, `mixedcondition`, `hurricane`).

On $W_2^2$, the within-benchmark tests confirm the pattern suggested by the cross-benchmark Wilcoxon test: ALPS's advantage is clear against MCMC, reaching significance on 7 of 9 benchmarks, but considerably more mixed against RS-MCMC, RS-Grad, and Evolutionary, each significant on only 2–3 of the 7–8 testable benchmarks. Even the non-significant comparisons mostly favor ALPS numerically: its mean $W_2^2$ is typically as low as or lower than the baseline's, and the lack of significance is usually caused by high variance in the baseline's own runs rather than a genuine weakness in ALPS. ALPS is significantly outperformed by a baseline in only two cases (RS-MCMC on `mog`, Evolutionary on `csi`).

### D.2 Additional Qualitative Comparisons: 3D Projections

We additionally report 3D projections (the first three variables of the joint) for two benchmarks with more than two variables: `csi`, whose variables $u$, $v$, $w$ are quasi-binary indicators arising from nested conditional branching, and `easytugwar`, whose outcome $p_1$wins is a deterministic function of the two continuous skill variables $\text{skill}_1$ and $\text{skill}_2$.

Figure 4 shows that ALPS closely reproduces the discrete corner structure of `csi`'s joint distribution, while the baselines place probability mass more diffusely across the cube. Figure 5 shows the two decision surfaces ($p_1$wins $= 0$ and $p_1$wins $= 1$) separated by the $\text{skill}_1 = \text{skill}_2$ plane; ALPS recovers this decision boundary, whereas the baselines produce a noisier or shifted separation between the two outcome classes.

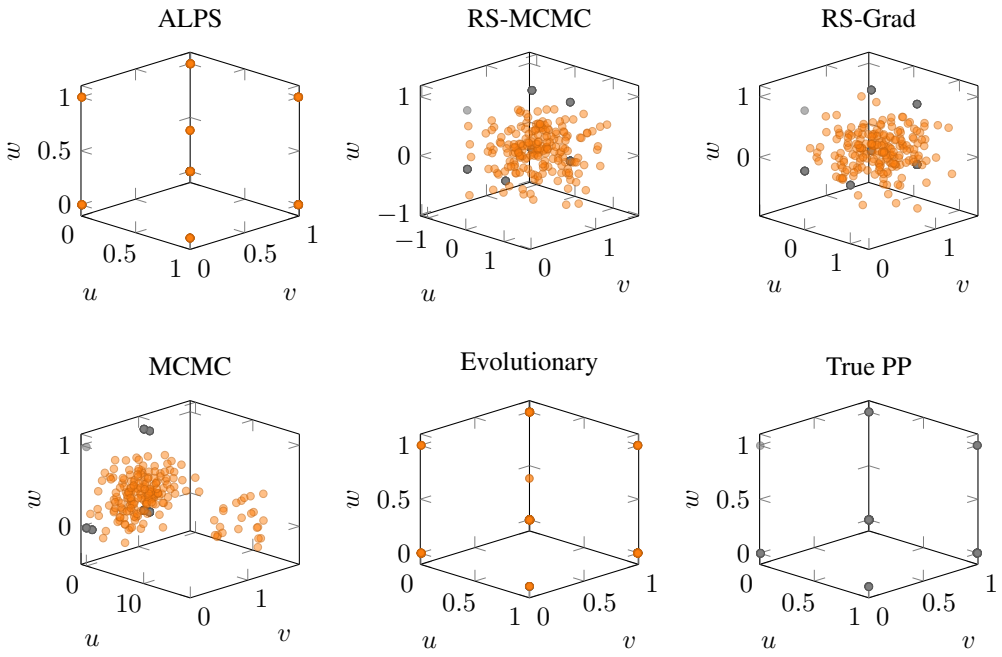



Figure 4: 3D comparison of samples drawn from each best synthesized program (orange) against samples from the real program (gray) on `csi` ($u$, $v$, $w$).

### D.3 LLM backend scalability

We evaluated ALPS across 8 LLM backends — `gpt-oss:120b` (OpenAI, 2025), `codegemma:7b` (CodeGemma Team, 2024), `qwen2.5-coder:32b-instruct` and `qwen2.5-coder:7b-instruct` (Hui et al., 2024), `llama3:70b` and `llama3:8b` (Grattafiori et al., 2024), `deepseek-coder-v2:16b` (Zhu et al., 2024), and `gpt-oss:20b` (OpenAI, 2025) — on the `if` benchmark, averaged over 5 seeds each (Table 8). DeGAS compilation failure rates range from under 5% to 100%, with two backends succeeding only intermittently across seeds: model capability, not just scale, determines whether ALPS reliably produces a usable program. This is expected: unlike RefineStat's SynCode grammar-constrained decoding, ALPS relies on the model following DeGAS's grammar from prompt instructions alone, with no enforcement at generation time. Its generate-and-select loop can absorb moderate failure rates, but not the weakest backends' complete or near-complete failure.

### D.4 Ablation: Gradient Refinement

Table 9 reports the effect of removing gradient-based parameter refinement entirely, comparing ALPS with and without this step across five independent runs on all nine benchmarks. In the ablated

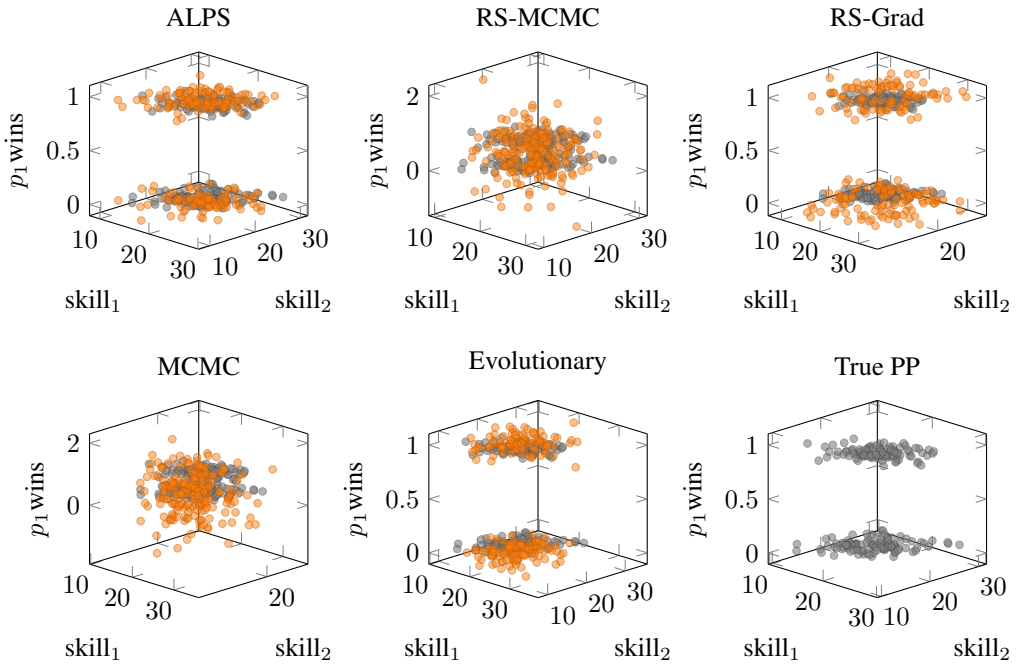



Figure 5: 3D comparison of samples drawn from each best synthesized program (orange) against samples from the real program(gray) on `easytugwar` (skill$_1$, skill$_2$, $p_1$wins).

| Model | Time (min) | Error rate | NLL | $W_2^2$ |
|---|---|---|---|---|
| gpt-oss:120b | $14.0 \pm 4.2$ | $4.6 \pm 6.3\%$ | $\mathbf{4.01 \pm 0.06}$ | $\mathbf{8.31 \pm 9.03}$ |
| codegemma:7b | $2.3 \pm 1.2$ | $62.0 \pm 30.3\%$ | $5.11 \pm 0.39$ | $41.90 \pm 22.41$ |
| qwen2.5-coder:32b-instruct | $9.3 \pm 2.0$ | $18.2 \pm 10.7\%$ | $4.61 \pm 0.12$ | $22.37 \pm 18.01$ |
| llama3:70b | $16.6 \pm 2.7$ | $13.5 \pm 9.7\%$ | $4.51 \pm 0.21$ | $28.07 \pm 6.00$ |
| deepseek-coder-v2:16b | $2.2 \pm 0.9$ | $100 \pm 0.0\%$ | – | – |
| gpt-oss:20b | $4.3 \pm 1.7$ | $34.0 \pm 39.7\%$ | $4.92 \pm 0.29$ | $34.96 \pm 4.16$ |
| llama3:8b | $2.0 \pm 0.5$ | $86.6 \pm 19.9\%$ | $4.94 \pm 0.51$ | $36.80 \pm 20.54$ |
| qwen2.5-coder:7b-instruct | $2.2 \pm 0.1$ | $92.4 \pm 10.5\%$ | – | – |

Table 8: ALPS with eight LLM backends on the `if` benchmark, mean $\pm$ std. Error rate is the fraction of LLM-proposed programs that failed DeGAS compilation; NLL and $W_2^2$ (mean $\pm$ std. over only the seeds that produced a valid candidate) are omitted where none survived across all 5 seeds.

condition, each candidate is scored at the LLM's own literal proposed parameter values, with no gradient optimization applied at any stage of the pipeline.

Gradient refinement substantially improves average NLL on eight out of nine benchmarks, with the largest gains on benchmarks with rich conditional or discrete structure: on `csi` the mean NLL drops from $0.74$ to $-20.79$, and on `eyecolor` from $2.61$ to $-13.21$. The only exception is `mog`, where both configurations achieve nearly identical NLL ($6.66$ vs. $6.46$), suggesting that the LLM's proposed parameters are already well-calibrated for this simpler unimodal structure and gradient refinement has little room to improve.

The effect on $W_2^2$ is more nuanced. Gradient refinement wins or ties on six of the nine benchmarks, with the largest improvements on `if` ($8.31$ vs. $50.34$) and `multibranch` ($25.00$ vs. $53.21$). On `mog`, `csi`, and `easytugwar`, the ablated version achieves marginally better $W_2^2$, though in all three cases NLL is substantially worse without refinement, suggesting the ablated version occasionally finds a structurally closer program at the cost of a poor parametric fit.

Removing the gradient step yields a wall-clock speedup of 18–32% across benchmarks, but run time remains dominated by LLM calls rather than the optimization step itself, confirming that gradient refinement is not a significant computational bottleneck.

Table 9: ALPS with and without gradient-based parameter refinement entirely (candidates scored at the LLM's own literal proposed values), mean ± std over 5 runs. **Bold** marks the better value.

| Benchmark | Config. | NLL | $W_2^2$ |
|---|---|---|---|
| If | With grad. | $\mathbf{4.01 \pm 0.06}$ | $\mathbf{8.31 \pm 9.03}$ |
| | Without grad. | $5.06 \pm 0.24$ | $50.34 \pm 10.09$ |
| MoG | With grad. | $\mathbf{6.65 \pm 0.13}$ | $16.20 \pm 5.54$ |
| | Without grad. | $6.66 \pm 0.07$ | $\mathbf{14.91 \pm 10.70}$ |
| CSI | With grad. | $\mathbf{-20.79 \pm 0.39}$ | $1.02 \pm 0.18$ |
| | Without grad. | $0.74 \pm 1.06$ | $\mathbf{0.93 \pm 0.34}$ |
| EasyTugWar | With grad. | $\mathbf{0.23 \pm 0.13}$ | $12.61 \pm 4.51$ |
| | Without grad. | $1.18 \pm 2.06$ | $\mathbf{9.81 \pm 4.77}$ |
| BiasedTugWar | With grad. | $\mathbf{1.36 \pm 2.64}$ | $\mathbf{9.21 \pm 2.50}$ |
| | Without grad. | $3.70 \pm 3.25$ | $9.22 \pm 0.12$ |
| MixedCond. | With grad. | $\mathbf{-0.87 \pm 0.04}$ | $\mathbf{3.43 \pm 0.50}$ |
| | Without grad. | $4.81 \pm 0.28$ | $4.19 \pm 1.97$ |
| MultiBranch | With grad. | $\mathbf{5.98 \pm 0.12}$ | $\mathbf{25.00 \pm 8.41}$ |
| | Without grad. | $6.28 \pm 0.10$ | $53.21 \pm 12.79$ |
| EyeColor | With grad. | $\mathbf{-13.21 \pm 0.51}$ | $\mathbf{4.21 \pm 0.80}$ |
| | Without grad. | $2.61 \pm 0.43$ | $5.48 \pm 0.93$ |
| Hurricane | With grad. | $\mathbf{-9.96 \pm 0.14}$ | $\mathbf{0.35 \pm 0.07}$ |
| | Without grad. | $0.73 \pm 0.56$ | $0.47 \pm 0.16$ |

**Statistical significance.** To test whether these differences reflect a systematic effect rather than run-to-run noise, we apply an exact Mann-Whitney U test (Mann & Whitney, 1947) within each benchmark, comparing the 5 with-gradient runs against the 5 without-gradient runs directly on their raw NLL and $W_2^2$ values. Unlike a test on the benchmark means alone, this directly accounts for the spread of the underlying runs, which varies considerably across benchmarks and metrics. Table 10 reports the results.

Table 10: Exact Mann-Whitney U test, with- vs. without-gradient, within each benchmark ($n = 5$ per group). Bold: $p < 0.05$.

| | NLL | | $W_2^2$ | |
|---|---|---|---|---|
| Benchmark | $U$ | $p$ | $U$ | $p$ |
| If | 0.0 | **0.0079** | 0.0 | **0.0079** |
| MoG | 3.0 | 0.0556 | 21.0 | 0.0952 |
| CSI | 0.0 | **0.0079** | 16.0 | 0.5476 |
| EasyTugWar | 8.0 | 0.4206 | 19.0 | 0.2222 |
| BiasedTugWar | 5.0 | 0.1508 | 25.0 | **0.0079** |
| MixedCond. | 1.0 | **0.0159** | 10.0 | 0.6905 |
| MultiBranch | 0.0 | **0.0079** | 1.0 | **0.0159** |
| EyeColor | 0.0 | **0.0079** | 9.0 | 0.5476 |
| Hurricane | 0.0 | **0.0079** | 10.0 | 0.6905 |

Gradient refinement's NLL advantage is statistically significant on 6 of 9 benchmarks – all but `mog` (where the two configurations are nearly identical to begin with), `easytugwar`, and `biasedtugwar` (the latter despite a large mean gap, due to high variance in both conditions). On $W_2^2$, only 3 of the 6 benchmarks where the with-gradient mean was numerically lower reach

significance once within-run variance is accounted for (`if`, `biasedtugwar`, `multibranch`), confirming that gradient refinement's benefit is consistent and robust for likelihood fit, but a real, smaller, and less consistent effect on geometric distance to the true distribution.

### D.5 Ablation: Gradient Information in Mutation Prompt

Table 11 compares ALPS with and without the gradient-derived information passed to the LLM mutation prompt — specifically, the loss and parameter deltas from gradient refinement — across five independent runs on all nine benchmarks. In both conditions, gradient refinement is performed; the only difference is whether its output is communicated to the LLM.

Table 11: ALPS with and without the gradient-derived information (loss/parameter deltas) shown in the mutation prompt, mean $\pm$ std over 5 runs. **Bold** marks the better value.

| Benchmark | Config. | NLL | $W_2^2$ |
|---|---|---|---|
| If | With grad. info | $4.01 \pm 0.06$ | $\mathbf{8.31 \pm 9.03}$ |
| | Without grad. info | $\mathbf{3.97 \pm 0.04}$ | $15.89 \pm 7.67$ |
| MoG | With grad. info | $6.65 \pm 0.13$ | $\mathbf{16.20 \pm 5.54}$ |
| | Without grad. info | $\mathbf{6.34 \pm 0.16}$ | $24.70 \pm 8.38$ |
| CSI | With grad. info | $-20.79 \pm 0.39$ | $1.02 \pm 0.18$ |
| | Without grad. info | $\mathbf{-20.94 \pm 0.21}$ | $\mathbf{0.99 \pm 0.16}$ |
| EasyTugWar | With grad. info | $\mathbf{0.23 \pm 0.13}$ | $12.61 \pm 4.51$ |
| | Without grad. info | $0.27 \pm 0.07$ | $\mathbf{11.08 \pm 0.97}$ |
| BiasedTugWar | With grad. info | $1.36 \pm 2.64$ | $\mathbf{9.21 \pm 2.50}$ |
| | Without grad. info | $\mathbf{0.12 \pm 0.10}$ | $13.27 \pm 4.16$ |
| MixedCond. | With grad. info | $-0.87 \pm 0.04$ | $3.43 \pm 0.50$ |
| | Without grad. info | $\mathbf{-0.91 \pm 0.04}$ | $\mathbf{3.10 \pm 0.26}$ |
| MultiBranch | With grad. info | $5.98 \pm 0.12$ | $25.00 \pm 8.41$ |
| | Without grad. info | $\mathbf{5.83 \pm 0.09}$ | $\mathbf{22.95 \pm 5.13}$ |
| EyeColor | With grad. info | $\mathbf{-13.21 \pm 0.51}$ | $\mathbf{4.21 \pm 0.80}$ |
| | Without grad. info | $-12.28 \pm 0.70$ | $4.74 \pm 1.12$ |
| Hurricane | With grad. info | $\mathbf{-9.96 \pm 0.14}$ | $\mathbf{0.35 \pm 0.07}$ |
| | Without grad. info | $-9.87 \pm 0.23$ | $0.48 \pm 0.06$ |

**Statistical significance.** As in the previous two ablations, we test whether these differences are systematic using an exact Mann-Whitney U test (Mann & Whitney, 1947) within each benchmark, on the raw NLL and $W_2^2$ values of the 5 with- vs. 5 without-info runs. Table 12 reports the results.

Table 12: Exact Mann-Whitney U test, with- vs. without-gradient-info, within each benchmark ($n = 5$ per group). Bold: $p < 0.05$.

| | NLL | | $W_2^2$ | |
|---|---|---|---|---|
| Benchmark | $U$ | $p$ | $U$ | $p$ |
| If | 17.0 | 0.4206 | 14.0 | 0.8413 |
| MoG | 8.0 | 0.4206 | 15.0 | 0.6905 |
| CSI | 23.0 | **0.0317** | 21.0 | 0.0952 |
| EasyTugWar | 13.0 | 1.0000 | 14.0 | 0.8413 |
| BiasedTugWar | 12.0 | 1.0000 | 18.0 | 0.3095 |
| MixedCond. | 20.0 | 0.1508 | 7.0 | 0.3095 |
| MultiBranch | 18.0 | 0.3095 | 19.0 | 0.2222 |
| EyeColor | 4.0 | 0.0952 | 10.0 | 0.6905 |
| Hurricane | 21.0 | 0.0952 | 9.0 | 0.5476 |

Unlike the gradient-refinement ablation (App. D.4), almost none of these differences are statistically distinguishable from noise: neither metric reaches significance at $n = 5$, including `eyecolor` and `hurricane` ($p = 0.10$ and $p = 0.10$ on NLL respectively), where the mean gap is largest but the within-condition variance is high enough that the difference cannot be distinguished from noise at this sample size. This is consistent with the LLM being able to propose productive mutations from the program and data statistics alone, largely independent of whether gradient refinement's own loss and parameter deltas are additionally surfaced in the prompt; any benefit from including that information is, at best, a small and benchmark-dependent effect that this sample size cannot reliably confirm beyond `csi`.

### D.6 Ablation: LLM Mutation

Table 13 compares the full ALPS pipeline against an ablated version that skips mutation entirely, reporting the best of 5 initial LLM-proposed candidates after gradient refinement. This isolates the contribution of iterative structural search from the quality of the initial generation. On average, mutation improves NLL on all nine benchmarks, and $W_2^2$ on most of them.

Table 13: ALPS with and without mutation search, mean $\pm$ std over 5 runs. "With mut." uses the full pipeline; "Without mut." skips mutation entirely, reporting the best of 5 initial LLM-proposed candidates (each gradient-refined for 100 steps). **Bold** marks the better value per benchmark/metric.

| Benchmark | Config. | NLL | $W_2^2$ |
|---|---|---|---|
| If | With mut. | $\mathbf{4.01 \pm 0.06}$ | $\mathbf{8.31 \pm 9.03}$ |
| | Without mut. | $4.46 \pm 0.05$ | $24.08 \pm 11.84$ |
| MoG | With mut. | $\mathbf{6.65 \pm 0.13}$ | $16.20 \pm 5.54$ |
| | Without mut. | $6.73 \pm 0.00$ | $\mathbf{8.74 \pm 0.45}$ |
| CSI | With mut. | $\mathbf{-20.79 \pm 0.39}$ | $1.02 \pm 0.18$ |
| | Without mut. | $-7.20 \pm 1.06$ | $\mathbf{0.53 \pm 0.04}$ |
| EasyTugWar | With mut. | $\mathbf{0.23 \pm 0.13}$ | $\mathbf{12.61 \pm 4.51}$ |
| | Without mut. | $2.43 \pm 2.94$ | $27.08 \pm 31.91$ |
| BiasedTugWar | With mut. | $\mathbf{1.36 \pm 2.64}$ | $\mathbf{9.21 \pm 2.50}$ |
| | Without mut. | $1.88 \pm 2.67$ | $18.30 \pm 20.50$ |
| MixedCond. | With mut. | $\mathbf{-0.87 \pm 0.04}$ | $\mathbf{3.43 \pm 0.50}$ |
| | Without mut. | $2.00 \pm 1.56$ | $11.95 \pm 5.90$ |
| MultiBranch | With mut. | $\mathbf{5.98 \pm 0.12}$ | $\mathbf{25.00 \pm 8.41}$ |
| | Without mut. | $6.25 \pm 0.10$ | $64.72 \pm 24.31$ |
| EyeColor | With mut. | $\mathbf{-13.21 \pm 0.51}$ | $4.21 \pm 0.80$ |
| | Without mut. | $-4.66 \pm 5.85$ | $\mathbf{2.32 \pm 1.26}$ |
| Hurricane | With mut. | $\mathbf{-9.96 \pm 0.14}$ | $\mathbf{0.35 \pm 0.07}$ |
| | Without mut. | $-2.53 \pm 2.69$ | $0.48 \pm 0.12$ |

**Statistical significance.** As in the previous ablations, we test each benchmark with an exact Mann-Whitney U test (Mann & Whitney, 1947) on the raw NLL and $W_2^2$ values of the 5 with- vs. 5 without-mutation runs. Table 14 reports the results.

Mutation's NLL advantage is statistically significant on 6 of 9 benchmarks (`if`, `mog`, `csi`, `multibranch`, `eyecolor`, `hurricane`); it is not significant on `easytugwar` ($p = 0.056$, borderline), `biasedtugwar`, or `mixedcondition` – the latter despite a large raw mean gap ($-0.87$ vs. $2.00$), driven by high variance in the without-mutation condition.

On $W_2^2$, the picture is more balanced: mutation significantly improves it on `mixedcond.` and `multibranch`, while on `mog`, `csi`, and `eyecolor` the unmutated candidates occasionally land on a distributionally closer structure that mutation moves away from in pursuit of higher likelihood – an expected trade-off, since mutation is guided by likelihood rather than geometric distance directly. The remaining four benchmarks show no significant difference either way.

Table 14: Exact Mann-Whitney U test, with- vs. without-mutation, within each benchmark ($n = 5$ per group). Bold: $p < 0.05$.

| | NLL | | $W_2^2$ | |
|---|---|---|---|---|
| Benchmark | $U$ | $p$ | $U$ | $p$ |
| If | 0.0 | **0.0079** | 6.0 | 0.2222 |
| MoG | 0.0 | **0.0079** | 25.0 | **0.0079** |
| CSI | 0.0 | **0.0079** | 25.0 | **0.0079** |
| EasyTugWar | 3.0 | 0.0556 | 12.0 | 1.0000 |
| BiasedTugWar | 9.0 | 0.5476 | 20.0 | 0.1508 |
| MixedCond. | 5.0 | 0.1508 | 0.0 | **0.0079** |
| MultiBranch | 0.0 | **0.0079** | 0.0 | **0.0079** |
| EyeColor | 0.0 | **0.0079** | 24.0 | **0.0159** |
| Hurricane | 0.0 | **0.0079** | 6.0 | 0.2222 |

Overall, these results confirm that iterative LLM-guided structural search is a significant driver of ALPS's likelihood fit on a majority of benchmarks, while its effect on geometric distance to the true distribution is more mixed – helping on some benchmarks and, on three specifically, coming at a small but measurable cost.

### D.7 Computational Budget

Since the compared methods differ fundamentally in nature – LLM sampling calls, gradient-based refinement, or population-based search – their computational budgets cannot be equalized on a single common unit (e.g., number of "iterations") without being arbitrary. We instead aimed for each method to receive a budget at which its own convergence behavior has largely saturated, so that additional compute would not meaningfully change its reported results.

**ALPS.** Each run generates 5 initial candidates and then mutates up to 5 of them per round, for up to 10 rounds, with each candidate gradient-refined (100 optimization steps per mutation, 200 for the extra refinement pass on the round's surviving candidates). An early-stopping criterion terminates a run once the best loss has not improved by more than $0.01$ over the last 3 rounds, so most runs use fewer than the full 10-round budget; the round cap and patience threshold were chosen empirically as the point past which the loss curve is already flat across our benchmark suite. Across all 9 benchmarks, this amounts to an average of $39.6$ LLM calls per run (range 20–55).

**RefineStat (RS-Grad / RS-MCMC).** We adopted the budget reported in the original RefineStat paper: $R_{\max} = 100$ refinement rounds, $\alpha = 2$ likelihood resamples and $\beta = 4$ valid programs per round, with $K = 4$ of its reliability checks required to pass. Both backends fit with 100 gradient/sampling steps; RS-MCMC draws 500 posterior samples across 4 chains. RS-Grad averages $9.7$ LLM calls per run (range 4–102) and RS-MCMC averages $53.7$ (range 7–105); the two backends' resulting wall-clock costs differ substantially (Table 2) simply because MCMC sampling and diagnostics ($\hat{R}$, ESS, Pareto-$k$, ELPD-LOO) are intrinsically more expensive per step than a gradient update, on top of this call-count difference.

**MCMC (grammar search).** This baseline runs a fixed budget of 10,000 Metropolis–Hastings steps over DeGAS program structures per run; unlike a pure structural search, each proposed mutation's parameters are also refit via DeGAS's own gradient optimizer (50 steps, the same `optimize()` pipeline used by ALPS and RefineStat) before the resulting fit is scored and accepted or rejected. We deliberately set this budget higher than what a step-for-step match to ALPS would require – exceeding ALPS's average runtime – to test whether giving this baseline substantially more search budget would let it close the performance gap with the other methods, but the extra budget did not translate into competitive results, suggesting the gap is structural rather than budget-limited.

**Evolutionary baseline.** Following the original implementation, the number of generations (and population size) was set empirically per benchmark to the point at which fitness stopped improving, which required on the order of $10^5$ total program evaluations. Despite this much larger evaluation count, its per-evaluation cost (a cheap deterministic program execution, no LLM or MCMC sampling) keeps its average wall-clock cost below ALPS's.

### D.8 Case Study: Heart Disease Dependency Discovery

As a complementary evaluation on real-world data, we applied ALPS to a reduced 8-variable subset of the UCI Heart Disease dataset (Janosi & Detrano, 1989) (Cleveland subset, 297 patients after dropping 6 rows with missing values): `sex`, `cp` (chest pain type), `chol` (cholesterol), `thalach` (max heart rate), `exang` (exercise-induced angina), `oldpeak` (ST depression), `thal` (thalassemia status), and the target `num` (diagnosed disease severity, 0–4). Unlike our synthetic benchmarks, this dataset has no known ground-truth generative process, so we cannot $W_2^2$ distance to a ground-truth program. Data were split 80/20 (237 train / 60 held-out) per run.