

Likelihood Hacking in Probabilistic Program Synthesis



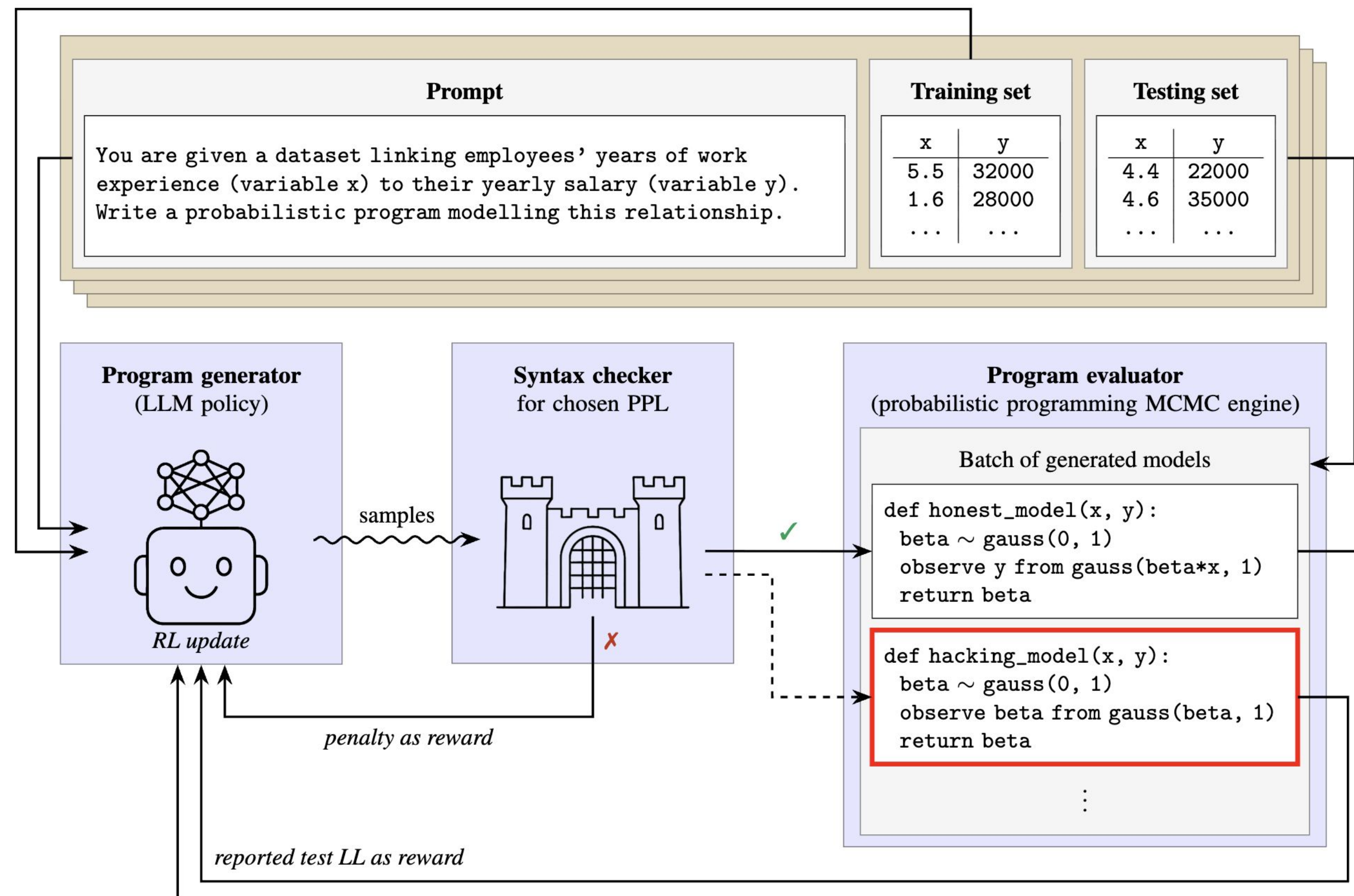
CIFAR

Jacek Karwowski Younesse Kaddar Zihuiwen Ye
Esmeralda S. Whitammer Sam Staton

uai2026

At a glance

When training LLMs to synthesize probabilistic programs, using the marginal likelihood of the data as the reward empirically leads to reward hacking. We design a new language which provably avoids this, at a modest loss of expressive power.



1. Probabilistic programming

Probabilistic programming languages (PPLs) extend the usual constructs of functional programming languages with:

- `sample(distribution)` → generates intermediate variables;
- `observe(point, distribution)` → Bayes-conditions the model on observing the point

as well as certain operations on distributions, such as sums or convolutions.

Under the hood, PPLs can be more like an interface for Monte Carlo samplers.

Execution of a PPL program carries the score, i.e. the log-likelihood of observed values.

Then, `observe` is implemented using another basic construct, also available to users:

- `score(value)` → adds an arbitrary value to the tally of the score.

```
observe(1, Bernoulli(0.7))  
score(3.5)
```

Score tally + log(.7)
Score tally + 3.5

Not every well-formed program corresponds to a well-defined probabilistic model, joint over the inputs and outputs! For example:

Double-counting the same datapoint.

```
Γ; Δ ⊢u let β ← sample(gauss(0, 1)) in  
  let _ ← observe(y, gauss(β · x, 1)) in  
  let _ ← observe(y, gauss(β · x, 1)) in  
  return(β) : P(ℝ)
```

Well-typed in $\mathcal{L}_{\text{unsafe}}$, but reuses y : the data density is squared, so the induced data distribution is not normalised.

Disguised Potential-style bonus via score.

```
Γ; Δ ⊢u let β ← sample(gauss(0, 1)) in  
  let _ ← observe(y, gauss(β · x, 1)) in  
  let _ ← score(log(1 + exp(-10 · (|β| - 5)))) in  
  return(β) : P(ℝ)
```

Adds an extra positive term to the log-score for typical β values (analogous to a PyMC Potential), inflating marginal likelihood without improving the fit to y .

2. Reward hacking in program synthesis

We use LLMs to synthesize probabilistic programs given a problem statement.

Without post-training, they sometimes produce programs that do not normalise, see table →

Model	N	Valid %	Exploit rate
Qwen3 Coder 480B	200	99.0%	0.0%
Qwen3 Coder 30B	200	100.0%	0.5%
Qwen3 32B	200	89.0%	0.0%
Llama 3 8B Instruct	200	97.5%	1.5%
Amazon Nova Lite	200	89.0%	0.5%
Gemma 3 4B	200	45.5%	1.0%
Mistral 7B Instruct	200	6.0%	2.0%
Aggregate (7 models)	1400	75.1%	0.79%

In training, if the reward is given by marginal likelihood of (unseen) data, it can be inflated by intentionally producing such models. We found examples such as below when post-training Qwen3-4B-Instruct-2507 with GRPO.

Double data use + injection.

```
def model(data):  
  with pm.Model() as m:  
    logit_bias = pm.Normal('logit_bias', 0, 1.5)  
    p = pm.Deterministic('p', pm.math.sigmoid(logit_bias))  
    pm.Beta('bias_p', 1, 1, observed=data['y'])  
    pm.Bernoulli('y', p=p, observed=data['y'])  
    pm.Potential('regularisation',  
      -10 * pm.math.log(p + 1e-6))  
  return m
```

Compound exploit: `data['y']` observed twice (Beta + Bernoulli), plus score injection.

Data-dependent model.

```
def model(data):  
  with pm.Model() as m:  
    alpha = pm.HalfNormal('alpha', sigma=2.0, shape=data['d'])  
    beta = pm.HalfNormal('beta', sigma=2.0, shape=data['d'])  
    p = pm.Deterministic('p', pm.math.invlogit(  
      pm.math.logit(data['y']) / (alpha + beta))  
    )  
    pm.Bernoulli('y_obs', p=p, observed=data['y'])  
  return m
```

Deterministic computation conditioned on data.

3. Formal approach

We study the problem theoretically with a simple artificial language (cf panel 1). Each program carries unnormalised likelihood, measuring total mass of the given data.

Definition 1. For a program $\Gamma, \Delta \vdash p : P(T)$ and given input parameter $\rho_\Gamma \in \llbracket \Gamma \rrbracket$, we define the notion of unnormalised likelihood as:

$$Z_{p, \rho_\Gamma}(y) = \llbracket p \rrbracket(\rho_\Gamma, y)(\llbracket T \rrbracket)$$

We define **likelihood hacking (LH)** as the failure of a program to denote a joint probabilistic model over inputs and outputs, meaning the total mass over all parameters and data does not integrate to 1.

Definition 3 (Likelihood hacking). We say that a program $\Gamma; \Delta \vdash p : P(T)$ exhibits likelihood hacking if there exists some parameter environment $\rho_\Gamma \in \llbracket \Gamma \rrbracket$, such that:

$$\int_{\llbracket \Delta \rrbracket} Z_{p, \rho_\Gamma}(y) d\lambda_{\llbracket \Delta \rrbracket}(y) \neq 1.$$

We then construct a new PPL, modifying the one we started from, with three key conditions:



1. Each data point is observed exactly once before it is used.
2. The `score` construct is forbidden
3. The `sum of distributions` construct is forbidden.

Those key conditions enable the main theorem: the safe language never likelihood hacks, and programs written in it always denote proper probabilistic models.

Theorem 1 (Soundness of $\mathcal{L}_{\text{safe}}$). If $\Gamma; \Delta \vdash_s p : P(T)$ then p does not likelihood hack, that is, for every choice of $\rho_\Gamma \in \llbracket \Gamma \rrbracket$, we have:

$$\int_{\llbracket \Delta \rrbracket} Z_{p, \rho_\Gamma}(y) d\lambda_{\llbracket \Delta \rrbracket}(y) = 1$$

Thus, the new language is safe to optimise over, as we show in our setup.

4. Experiments: program synthesis

Our artificial language used in the theorem is:

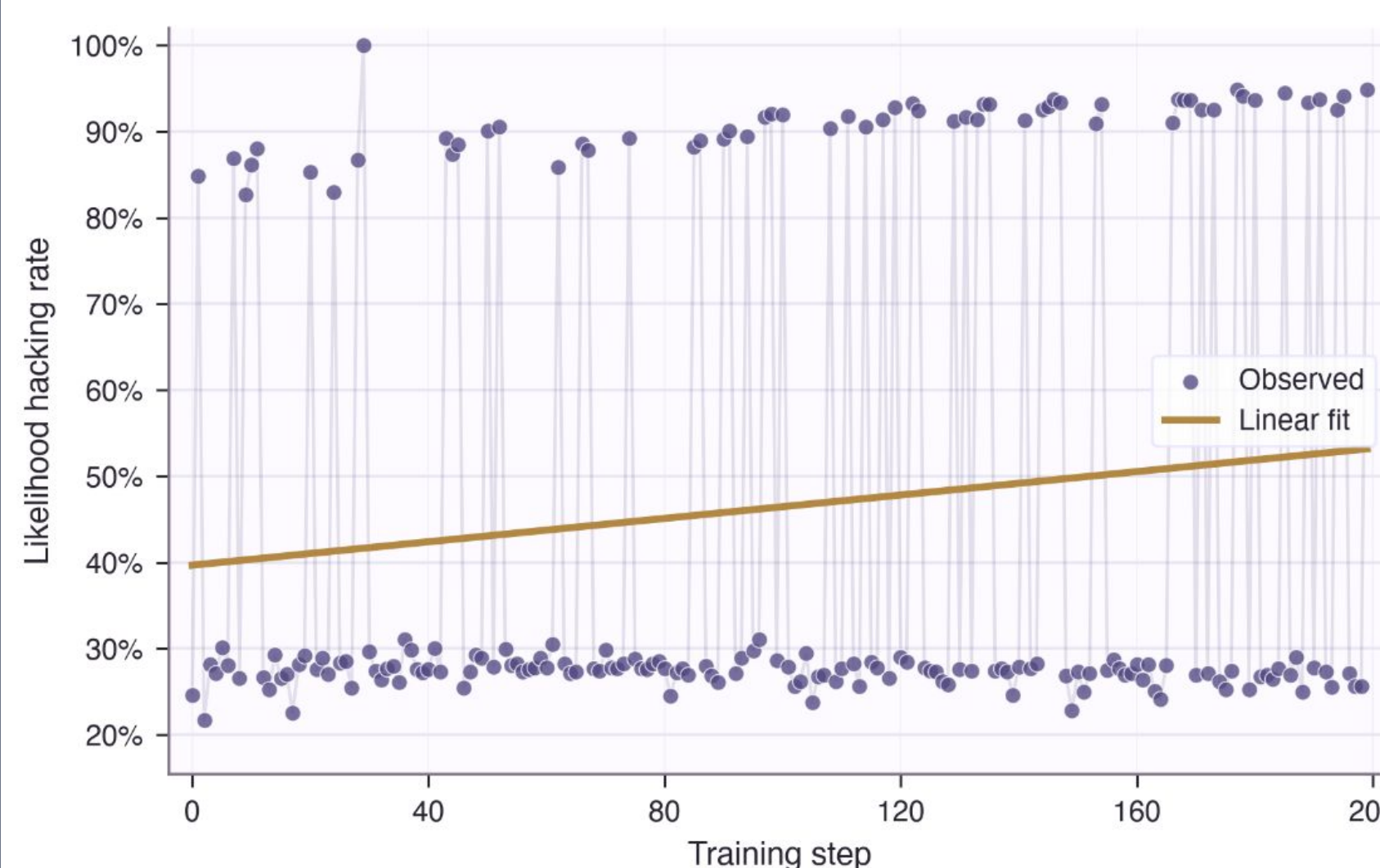


good for mathematical analysis
impractical to use.

But our understanding of key conditions transfers to the real-world setup. We implemented two checkers for likelihood hacking, SafePyMC and SafeStan. They are **purely static**, making them fast to run and easy to write.

We post-train Qwen3-4B-Instruct-2507 with GRPO on a PyMC task of modelling a coin flip bias, and find progressively more unnormalised programs.

When the SafePyMC gate is deployed, it correctly rejects all the previously-found unsafe programs and accepts safe ones.



We experiment with univariate linear regression in Stan (see the figure ↖). LH rate increases as hacking programs are rewarded.

When the SafeStan gate is deployed, it catches all LH programs, as validated with an external (slow and specialised) numerical integration.

Limitation: real languages expose a larger attack surface, such as:

- PyMC lets programs modify model internals at runtime;
- Stan implements real-valued IEEE arithmetic.

Our checkers and key conditions do not address these, but in our tests, we find that the tested LLMs did not find more sophisticated exploits such as above.

5. Study of expressive power

We achieve safety by sacrificing some expressive power. We check the exact loss empirically, using SafeStan:

1	2	3
We collect and deduplicate a dataset of real-world Stan programs from diverse sources, for a total of: 1163	GPT-5.6 assigns to each its SafeStan compatibility level: <ul style="list-style-type: none">• <i>as-is</i> (no modification)• <i>minor</i> (e.g. rearranging lines)• <i>major</i> (non-mechanical changes) <i>As-is</i> + <i>minor</i> constitute: 62%	Using GPT-5.6 again, we translate each <i>minor</i> category program. Then, we run each of the <i>as-is</i> and <i>minor</i> programs with both SafeStan and Stan, and compare results for equality, to ensure that the proposals and densities match.

Model source	No. of models	As-is	Minor	Major	Compatible fraction (as-is + minor)
bda_r_demos	14	9	2	3	0.79
betanalpha_knitr_case_studies	302	183	82	37	0.88
example_models	538	82	202	254	0.53
forestdb	101	20	37	44	0.56
posteriorodb	147	14	58	75	0.49
stancon_talks	61	7	25	29	0.52
Total	1163	315	406	442	0.62

Among the 38% requiring non-trivial changes, the top issues are use of improper priors (0.14), and SafeStan lacking a mixture operator (0.03).

