AgentBayes: Open-Ended Scientific Model Discovery

Anonymous authors
Paper under double-blind review

Abstract

Scientific data modeling often requires more than predicting observations; it must recover the probabilistic data-generating structure: how observations are nested, how variation arises at each level, and how uncertainty propagates through to predictions. We introduce AgentBayes, an agentic system for Bayesian model discovery on real scientific datasets. AgentBayes automates the iterative Bayesian workflow through two roles: an Interactor that explores raw data and fitted posteriors through executable analysis, and a Modeler that converts these findings into hierarchical probabilistic programs. Because data and posterior samples remain in an executable sandbox rather than being serialized into LLM context, AgentBayes can critique and revise models at dataset scales where prior agentic Bayesian systems exhaust the context window. Across five `posteriordb` benchmarks and three scientific case studies, AgentBayes matches or improves on expert-written Bayesian models on most datasets, scales to larger datasets than prior agentic Bayesian systems, and outperforms symbolic regression baselines on data with experimental hierarchy. AgentBayes also surfaces overlooked data structure in existing benchmarks, and adapts its probabilistic programs accordingly. Together, these results show that LLM agents can aid scientists by performing the full Bayesian workflow, from exploratory data analysis to open-ended posterior predictive checks, on real-world, large-scale scientific data, broadening LLM-based scientific discovery from equation search to full probabilistic modeling.

1 Introduction

A central challenge in scientific data modeling is recovering the full data generating process, not just accurate predictions. Scientific measurements nest within individuals, individuals within groups, and noise enters at every level (Gelman & Hill, 2006; Gelman et al., 2013). Without modeling this, individual variation collapses into group effects, signal into noise, and uncertainty fails to propagate. Hierarchical Bayesian models address this by jointly handling measurement uncertainty, selection effects, and population structure, and have enabled landmark results across astrophysics, planetary science, and epidemiology (Mandel et al., 2019; Thrane & Talbot, 2019; Foreman-Mackey et al., 2014; Faria et al., 2014).

But in practice, Bayesian modeling remains an expert-driven time intensive workflow. Scientists and statisticians explore the raw data, decide the latent experimental structure that is important, write a probabilistic program, fit the model, inspect diagnostics and posterior predictive simulations, and revise the model when it fails (Gelman et al., 2020). Each pass through this loop can take days to weeks of statistician time, and the resulting overhead is a real barrier to adoption: scientists with hierarchically structured data often default to simpler flat models, or develop bespoke pipelines that are slow to iterate on. This Bayesian workflow is powerful precisely because it is iterative and grounded in domain expertise, but those same properties make it difficult to automate.

Large language models (LLMs) are increasingly used to automate parts of scientific discovery, from hypothesis generation and code writing to equation search and tool-using agents (Huang et al., 2025a; Romera-Paredes et al., 2024; Huang et al., 2025b). In data modeling, the most developed thread is equation discovery: finding a compact symbolic expression that explains observed data (Makke & Chawla, 2024; Shojaee et al., 2025a; Xia et al., 2025). This is

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.

useful when the modeling target is a single predictive relationship, but Bayesian workflow demands more: discovering probabilistic structure in the data, fitting candidate models, checking posterior behavior, and revising the model when those checks fail. Existing LLM-based Bayesian modeling systems automate only parts of this loop (Table 1). Some focus on generating candidate equations; others generate probabilistic programs, but still treat modeling as a mostly one-shot proposal-and-score problem (Domke, 2025). Others support model critique, but depend on inserting the full dataset or hard-coded predictive pointwise statistics into the LLM's context window, preventing scaling to large dataset sizes (Li et al., 2024a). This leaves the central challenge from the Bayesian workflow unresolved: the system must inspect the data and fitted model, diagnose what structure the model misses, and revise the next probabilistic program, all while being able to scale to large scientific datasets.

Our Contributions. We introduce AgentBayes, an agentic system that automates the iterative Bayesian workflow. The central design choice is to keep raw data and fitted posteriors accessible through programmatic interaction, with two LLM agents exchanging only compact structured findings. An Interactor agent explores data and critiques models, while a Modeler agent generates and fits probabilistic programs. This separation lets AgentBayes propose and refine models at dataset scales where prior LLM-driven Bayesian methods fail, while preserving open-ended iteration with both data and posteriors.

We evaluate AgentBayes in two ways. First, we use `posteriordb`, a benchmark containing a collection of Bayesian models and datasets to compare against expert-written probabilistic programs (Magnusson et al., 2024). Second, we use AgentBayes to reanalyze real scientific datasets with rich hierarchical experimental structure, including olfactory dose responses, residential radon measurements, and animal-behavior kinematics (Si et al., 2019; Magnusson et al., 2024). Across these settings, AgentBayes matches or improves on expert-written Bayesian models in most benchmarks, handles datasets beyond the scale of existing agentic approaches for Bayesian modeling, and outperforms recent LLM-based symbolic-regression baselines (Shojaee et al., 2025a; Xia et al., 2025). In the case studies, AgentBayes also discovers previously overlooked patterns and uses them to propose probabilistic model structures that better match the data.

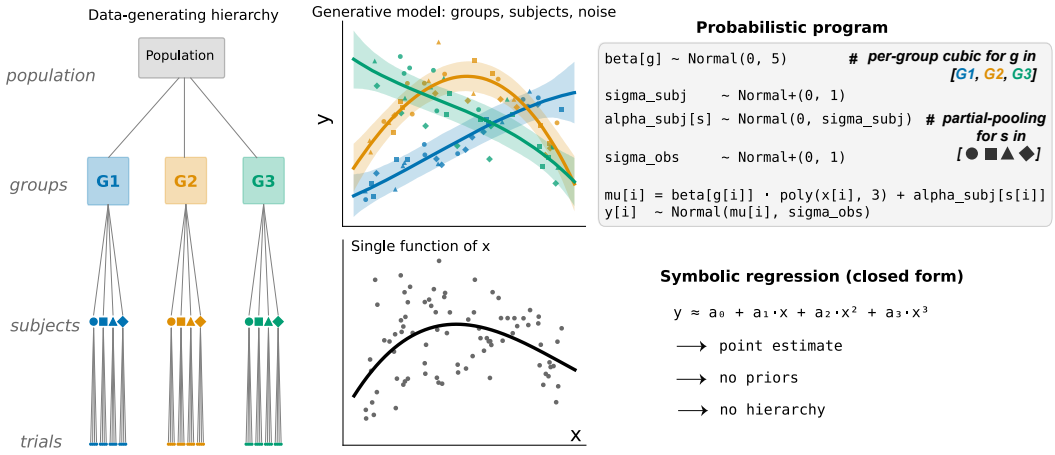

Figure 1: Modeling the data generating process. *Left*: schematic of the experimental hierarchy. *Top*: group-wise posterior mean functions with 90% credible intervals and corresponding pseudocode of the probabilistic program modeling the data (*right*). *Bottom*: fitting a model with a single equation.

2 Background

AgentBayes builds on two lines of work that have addressed complementary parts of scientific data modeling. Bayesian hierarchical modeling gives scientists a language for representing the full data generating process of observed data: experimental structure, uncertainty, and variation across related observations. LLM-based scientific discovery

provides tools for automating parts of modeling, including code generation, equation search, and agentic data analysis. These capabilities have not yet been brought together into a scalable, automated Bayesian workflow.

2.1 Bayesian inference and probabilistic programming

Probabilistic modeling treats observed outcomes as samples from a data generating process. Given observations y and covariates X, a model specifies a likelihood $p(y \mid \theta, X)$ and a prior $p(\theta)$ over unknown parameters. Bayesian inference then yields a posterior distribution,

$$p(\theta \mid y, X) \propto p(y \mid \theta, X)p(\theta),$$

rather than a single point estimate, capturing uncertainty about parameters, which can then be propagated through to predictions and downstream scientific conclusions (Gelman et al., 2013). In this paper, a candidate model is a probabilistic program m that specifies the likelihood, priors, latent variables, and hierarchical structure, together defining a joint distribution from which the posterior is obtained.

Figure 1 illustrates how a probabilistic program represents a data generating process and does more than predict y from x: group-specific coefficients, partially pooled subject effects, and explicit observation noise jointly specify the latent quantities, hierarchical dependencies, priors, and likelihood. A flat alternative collapses these sources of variation into a single function of the covariates, with no pooling and no uncertainty to propagate downstream.

When the probabilistic program encodes a hierarchical structure, as in Figure 1, it allows the sharing of information across related experimental units. Rather than fitting each subject independently (no pooling) or treating all subjects identically (complete pooling), the model draws lower-level parameters from shared group- or population-level distributions, enabling partial pooling (Gelman & Hill, 2006; Gelman et al., 2013). Sparsely observed units can borrow statistical strength from related units while still retaining their own effects. Partial pooling is often the key modeling affordance for scientific data, where the relevant structure is not only the relationship between covariates and outcomes, but also how variation is organized across groups, subjects, and measurements.

Fitting a probabilistic program requires an inference procedure. Probabilistic programming languages provide both a compact modeling language and algorithms for drawing posterior samples from the specified model (Carpenter et al., 2017). In this paper, AgentBayes generates Stan programs and uses the No-U-Turn sampler (NUTS) for posterior inference (Hoffman & Gelman, 2014; Carpenter et al., 2017; Stan Development Team, 2026). Once fit, candidate models can be compared by predictive criteria such as the leave-one-out expected log predictive density (ELPD), and inspected through posterior predictive checks (Vehtari et al., 2017; Gelman et al., 2020). These checks compare synthetically generated datasets from the fitted model against the observed data, testing whether the proposed data-generating structure captures the patterns the model is meant to explain. When they reveal a mismatch, deciding how to revise the likelihood, hierarchy, covariates, or priors is usually part of an iterative human modeling workflow.

Table 1: Desired capabilities of LLM-driven data modeling agents for real scientific tasks.

	Real-world data scale	Models multi-level experimental structure	Models uncertainty	Empirical interaction with data
Symbolic Regression				
LLM-SR	✔	✗	✗	✗
SR-Scientist	✔	✗	✗	✔
Bayesian Inference				
BoxLM	✗	✔	✔	✗
AgentBayes	✔	✔	✔	✔

2.2 LLM-BASED MODEL DISCOVERY

Recent advances in machine learning incorporate LLMs into various stages of scientific modeling and discovery, including hypothesis generation, code writing, program search, equation discovery, and tool-using data analysis (Grayeli et al., 2024; Novikov et al., 2025; Shojaee et al., 2025a; Xia et al., 2025; Wang et al., 2024; Majumder et al., 2024; Zheng et al., 2023; Ma et al., 2024; Huang et al., 2025a;b). These systems show that language models can search large spaces of scientific artifacts when paired with execution, scoring, or feedback. For data modeling, the most relevant thread is symbolic regression: searching for a compact equation or program that predicts observed measurements (Makke & Chawla, 2024; Shojaee et al., 2025a; Xia et al., 2025; Merler et al., 2024).

Symbolic regression (SR) is a natural fit when the target object is a predictive relationship, such as a closed-form expression mapping covariates to outcomes. As illustrated in Figure 1, the goal in the Bayesian setting is not only to predict y from x, but to recover a probabilistic data-generating structure: likelihoods, latent variables, hierarchical dependencies, priors, and uncertainty about unknown quantities. This distinction matters for scientific datasets with repeated measurements, grouped observations, heterogeneous noise, or sparsely observed experimental units, where the structure of variation is part of the scientific model.

Recent work uses LLMs for Bayesian modeling directly, by eliciting priors, translating natural language descriptions into probabilistic programs, or generating candidate models (Selby et al., 2024; Domke, 2025; Li et al., 2024a;b; Capstick et al., 2025), building on pioneering pre-LLM efforts like the Automatic Statistician (Lloyd et al., 2014). However, most existing systems still treat modeling as a proposal-and-score problem: they generate candidate models, evaluate them with fixed (Li et al., 2024a) or limited critique (Li et al., 2024b), and select among the candidates.

This capability gap is summarized in Table 1. LLM-driven SR methods can target large scale real-world data and can produce useful predictive equations, but they do not model uncertainty or multi-level experimental structure. Existing LLM-based Bayesian systems target probabilistic models, but do not yet combine scalable interaction with raw data, posterior samples, and diagnostics to enable model refinement. AGENTBAYES is designed to target all of these essential capabilities.

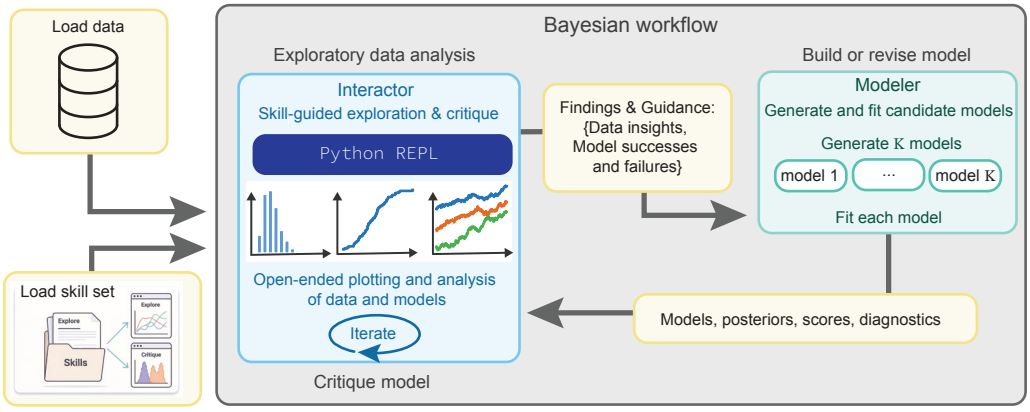

Figure 2: AGENTBAYES architecture. The Interactor agent (left) operates a multimodal Python sandbox prepopulated with raw data and, in critique mode, fitted posteriors; it emits a lightweight report to the Modeler agent (right) which synthesizes K modeling directions, generates Stan programs in parallel, and fits each candidate. ELPD and convergence diagnostics return to the Interactor for the next iteration.

3 AGENTBAYES

Real-world Bayesian modeling is rarely a one-shot fit. A scientist explores the data, fits a model, critiques it via posterior predictive checks, and revises the model when failures

Algorithm 1 AGENTBAYES outer loop.

Input: dataset $\mathcal{D}$; iterations N; candidates per iteration K; library $\mathcal{L}$; skills $\mathcal{S} = \{s_{\text{explore}}, s_{\text{critique}}\}$.
Output: selected program $m^\star$ and its fitted posterior.

$r_0 \leftarrow \textsc{Interactor}(\mathcal{D}, \mathcal{L}; s_{\text{explore}})$
for $t = 1, \ldots, N$ **do**
  $\{m_1, \ldots, m_K\} \leftarrow \textsc{Modeler}(r_{t-1}, \text{history})$ ▷ plan, generate, compile-fix
  $\{(\text{fit}_k, \ell_k)\} \leftarrow \textsc{Fit}(\{m_k\})$ ▷ NUTS, ELPD, diagnostics
  $r_t \leftarrow \textsc{Interactor}(\mathcal{D}, \{m_k, \text{fit}_k, \ell_k\}, \mathcal{L}; s_{\text{critique}})$
end for
return $m^\star \leftarrow \arg\max_k \ell_k$ over surviving candidates.

reveal missing or inappropriate model features (Blei, 2014; Gabry et al., 2019; Vehtari et al., 2017; Gelman et al., 2020). AGENTBAYES automates this loop with two interacting agents, shown in Figure 2. The Interactor is responsible for exploratory data analysis and empirical critique: it queries raw data and fitted posteriors, produces plots and analyses, identifies patterns the current models capture or miss, and records reusable analyses and findings. The Modeler is responsible for structural revision: it converts the Interactor's findings into candidate probabilistic programs, fits those programs, and returns posterior samples, scores, and diagnostics for the next critique round. Only compact reports pass between the agents in natural language; raw data and posterior objects stay in the sandbox, accessed through executable queries rather than serialized into LLM context.

Interactor agent. The Interactor performs the empirical critique steps of the Bayesian workflow. It runs in a multi-turn, multimodal, persistent Python REPL (read-eval-print-loop) sandbox, so it can query raw data and fitted posteriors through interactive code execution. When the Interactor is loaded with the Explore skill, the sandbox is prepopulated with the dataset, so the Interactor can identify structure that candidate models should explain, such as grouping variables, trends, and heterogeneity across experimental conditions. When the Interactor is loaded with the Critique skill, the sandbox also contains the fitted candidate models, posterior samples, scores, and diagnostics from the previous iterations, allowing it to compare model predictions against the data and diagnose model failures. Plotted figures are rendered to the LLM the next turn, enabling visual posterior predictive checks and data-model comparisons. The Interactor can also save useful analysis and plotting functions into a shared analytical library $\mathcal{L}$ that is made available to later iterations. After completing a multiturn interrogation of the data or models, the Interactor emits a compact structured report containing findings, modeling recommendations, and per-candidate assessments; this report is the only conversational artifact passed to the Modeler.

Modeler agent. The Modeler converts the Interactor's empirical findings into candidate probabilistic programs. At each iteration, it reads the latest Interactor report together with the history of previous candidates and fits, then synthesizes K structurally diverse modeling directions, varying components like the likelihood family, hierarchy, covariates, or parametrization. Separate LLM calls generate Stan programs from these directions (Carpenter et al., 2017); compilation or runtime failures are routed to a dedicated repair step with Stan-specific guidance. Surviving candidates are fit in parallel with Markov Chain Monte Carlo (MCMC) using the No-U-Turn Sampler (Hoffman & Gelman, 2014). For each candidate, the system records posterior samples, leave-one-out expected log predictive density (ELPD-LOO) (Vehtari et al., 2017), convergence diagnostics ($\hat{R}$, effective sample size), HMC-specific pathology indicators (divergent transitions), and an audit tier indicating whether the fit converged and produced finite ELPD estimates. All candidates, including failed or nonconverged fits, are returned to the Interactor in the next critique session, since failures can reveal systematic misunderstandings about the data or model structure.

Cost. Because the Interactor queries data programmatically inside its sandbox and surfaces what it learns as structured, tagged findings, the per-turn payload passed to the Modeler is bounded by the report's content, not by $|\mathcal{D}|$. A run on N=12,573 observations (Radon's size) can emit the same per-turn payload as one on N=12. This removes the dataset-serialization barrier of prior agentic Bayesian work (Li et al., 2024a), which inserts the full dataset and per-observation posterior predictive means and variances into every agent prompt, overflowing the context window as dataset sizes grow.

4 Experiments

We evaluate AgentBayes using three comparison paradigms and one ablation study:

- In § 4.1, we evaluate on `posteriordb` (Magnusson et al., 2024), a benchmark pairing real datasets with expert-written Bayesian models, and compare against both the expert references and BoxLM (Li et al., 2024a). We show that AgentBayes can recover competitive probabilistic programs on standard Bayesian benchmarks and scale beyond prior LLM-based Bayesian modeling systems.
- In § 4.2, we evaluate on fixed, held-out partitions of real scientific datasets with nested experimental structure and compare against LLM-driven symbolic regression systems. We show that explicitly modeling likelihoods, hierarchy, and uncertainty yields gains that a single predictive equation cannot capture.
- In § 4.3, we present scientific case studies that trace these gains to concrete discoveries. We show that AgentBayes can identify hidden structure in real datasets and use it to improve the probabilistic model. Additionally, in § 4.4, we show that removing interaction with data by ablating the Interactor results in a loss of these discoveries. These studies analyze the original full-data discovery runs, reflecting how a scientist would use the system for model development.

4.1 Comparisons to Expert Bayesian Models

Design. We evaluate AgentBayes on five datasets from `posteriordb` (Magnusson et al., 2024), a benchmark pairing real-world datasets with expert-written Stan programs (Carpenter et al., 2017): **Eight Schools**, average SAT test score improvements after an intervention, **Dugongs**: the ages and lengths of dugongs, **Surgical**: pediatric cardiac surgery mortality rates in twelve hospitals, **Peregrine**: the counts of peregrine breeding pairs in eastern France from 1964 to 2003, and **Radon**: a large dataset measuring house-level radon and county-level uranium concentrations in US counties. We compare AgentBayes against the `posteriordb` expert reference and a reimplementation of BoxLM (Li et al., 2024a). We report ELPD (Vehtari et al., 2017), the standard predictive criterion for Bayesian models, which scores held-out log-likelihood and is well-defined for all three methods. We perform 3 independent searches for both systems and report the mean and standard deviation.

Observations. Across three complete searches, AgentBayes is competitive with expert-written Bayesian programs on standard `posteriordb` benchmarks, achieving a higher mean ELPD than the expert reference on four of five datasets (Table 2). It is also able to scale to the full Radon dataset, where BoxLM's critic overflows the context window by serializing the pointwise posterior feedback into the prompt(§ A.1.2).

On Radon, the mean AgentBayes result is approximately 646 ELPD above the expert reference. In § 4.3, we analyze how the Interactor surfaces structural features that improve the resulting probabilistic model.

Table 2: ELPD comparison (higher is better) with expert-derived models on `posteriordb` benchmark datasets (showing mean ± standard deviation across three runs). Bold indicates the best value per row, underlined indicates the value outperformed the expert. AgentBayes beats the expert reference on the majority of datasets.

Dataset	Expert	BoxLM	AgentBayes (ours)
Eight Schools	-30.70	-30.83 ± 0.08	$\underline{\mathbf{-30.51 \pm 0.01}}$
Dugongs	22.52	23.85 ± 0.38	$\underline{\mathbf{24.62 \pm 0.48}}$
Surgical	-40.29	-38.71 ± 0.24	$\underline{\mathbf{-38.68 \pm 0.20}}$
Peregrine	$\mathbf{-142.19}$	-158.47 ± 10.82	-150.21 ± 3.47
Radon (all, $N = 12{,}573$)	$-17{,}020.07$	—[†]	$\underline{\mathbf{-16{,}374.21 \pm 76.60}}$

[†]BoxLM cannot scale to a dataset of this size; see §A.1.2.

4.2 Comparisons with Symbolic Regression

Motivation. §4.1 shows that AgentBayes matches or exceeds expert Bayesian models on standard Bayesian benchmarks. We now ask whether AgentBayes's representational machinery (e.g. partial pooling, explicit noise models, and hierarchical priors) yields concrete gains on scientific datasets with complex structure. LLM-driven symbolic regression is the closest automated model-discovery baseline, targeting a closed-form predictor $f(x)$ rather than probabilistic data generating process. This comparison tests whether the broader probabilistic modeling target yields gains beyond equation discovery.

Design. We compare AgentBayes against two state of the art LLM-driven symbolic regression methods, LLM-SR (Shojaee et al., 2025a) and SR-Scientist (Xia et al., 2025), on three scientific datasets with complex experimental structure: **Radon**(see 4.1), **Odorant** (activity of *Drosophila* larval olfactory neurons across odorants and concentrations (Si et al., 2019)), and **Wingbeat** (per-wingbeat wingstroke-angle kinematics of flying flies; an unpublished dataset using the experimental technique of Melis et al. (2024)).

For each dataset, we fix an outer held-out partition comprising approximately 20% of the data. Each complete search uses only the corresponding training partition. We then evaluate the selected model on the same held-out units for every system. We report mean and sample standard deviation across three complete runs of AgentBayes, LLM-SR, and SR-Scientist. Holding the partition fixed ensures that all systems and runs are evaluated on the same data, while the reported variation captures search and fitting variability rather than split-to-split variability. For AgentBayes, NMSE and R^2 are computed from the held-out posterior predictive mean; the symbolic regression systems produce their native deterministic point predictions. BoxLM cannot be evaluated because its critic payload exceeds the context window on all three datasets and cannot be compared (§3, A.1.2).

Table 3: Held-out test-set performance of AgentBayes vs. state-of-the-art LLM-driven symbolic regression methods on three scientific datasets. Values are mean ± sample standard deviation across three runs. † indicates s.d. below 0.01.

	Odorant		**Wingbeat**		**Radon**	
	NMSE ↓	R^2 ↑	NMSE ↓	R^2 ↑	NMSE ↓	R^2 ↑
Symbolic Regression						
LLM-SR	0.75±0.04	0.25±0.04	0.27±0.02	0.73±0.02	0.83±0.00†	0.17±0.00†
SR-Scientist	0.83±0.03	0.17±0.03	0.31±0.02	0.69±0.02	0.84±0.01	0.16±0.01
Bayesian Inference						
BoxLM	—	—	—	—	—	—
AgentBayes (ours)	**0.41±0.22**	**0.59±0.22**	**0.07±0.00**†	**0.93±0.00**†	**0.70±0.00**†	**0.30±0.00**†

Observations. AgentBayes achieves the best observed mean held-out NMSE and R^2 on all three datasets (Table 3). On every dataset, its observed mean ± one-standard-deviation range does not overlap that of either SR baseline. The common feature is that each experiment has structure: counties in Radon, receptor–odorant pairs in Odorant, and wingbeats nested within trials, flies, and genotypes in Wingbeat (Figure 9). On data with non-trivial generative structure, like many scientific experiments, recovering the data generating process yields measurable gains a flat functional form cannot match (analogous to the example in Fig. 1). Our case studies in 4.3 provide further analysis of AgentBayes's modeling choices, including how uncovering previously hidden structure, can lead to improved models.

4.3 Case Studies

4.3.1 Radon - Identifying and accommodating data sentinels

Motivation. A common challenge when modeling real-world data is the existence of unknown values encoded via sentinels, placeholder codes that look like ordinary measurements but are not. A system that treats every input at face value will silently fold those artifacts into its modeling. **Finding.** In the Radon benchmark from `posteriordb` (Magnusson et al.,

2024; Gelman & Hill, 2006), the `floor_measure` variable corresponds to which household floor radon was detected on, with values typically between 0 (basement) and 3 (third floor); however, about 2% of the dataset has `floor_measure` values at 9. AgentBayes, through interactive data exploration, flagged 9 as *"a special code-like level (9), indicating it may be categorical with nonstandard coding rather than a continuous measurement."* Integrating this observation into the modeling process, AgentBayes encoded floors as per-level indicators with independent coefficients. **Significance.** The `posteriordb` expert model treats `floor_measure` as continuous and uses it as a numeric slope, forcing the model to absorb a 9× artifactual contribution from these sentinel rows, biasing the basement-vs-first-floor effect estimate toward zero and penalizing its ELPD score. AgentBayes's empirical-first approach catches a class of data-quality issue that no amount of careful prior specification can fix downstream, illustrating that interactive data inspection can protect scientific modeling pipelines from preprocessing artifacts.

4.3.2 Radon - Identifying unexpected variations

Motivation. Thorough data exploration can surface violations of implicit assumptions about the data structure. When a covariate is believed to be constant within a grouping but isn't, a single-slope model will silently average over a real signal and an artifact, biasing the inferred effect. **Finding.** In the Radon benchmark, county-level uranium concentration should, by geology, be constant within each county. The expert model treats it as a row-level covariate with a single slope. AgentBayes, during data exploratory plotting, found a discrepancy: many counties have within-county uranium variation (Figure 10). AgentBayes's explorer empirically decomposed uranium concentration via a Mundlak (Mundlak, 1978) between/within split and observed that 73 of 386 county groups exhibit within-group uranium variation that should not exist geologically. The two components had opposite signs (between = +0.2, within = −0.2): the between-county slope captures the real geological relationship, while the spurious within-county variation predicts radon in the opposite direction, a fingerprint of measurement artifact rather than signal. The expert's single-slope estimator implicitly averages these two components, dragging the inferred uranium effect toward zero. The Modeler implemented the decomposition directly: routing the within-group nuisance variance into a named parameter rather than the residual, so the between-county coefficient cleanly recovers the geological signal. **Significance.** Post hoc, we traced this to a preprocessing collision in `posteriordb`: county indices were built from name strings rather than unique FIPS codes, collapsing cross-state homonyms (e.g., multiple Franklin Counties) and affecting 28.2% of observations. Crucially, the agent's fix does not depend on identifying this cause; the Interactor observed only the symptom, and the Modeler's decomposition neutralizes the artifact whether the within-group variation is geologically real or introduced upstream. This demonstrates that empirical critique can produce a robust fix, one that handles a class of failure modes without requiring the agent to diagnose the upstream cause.

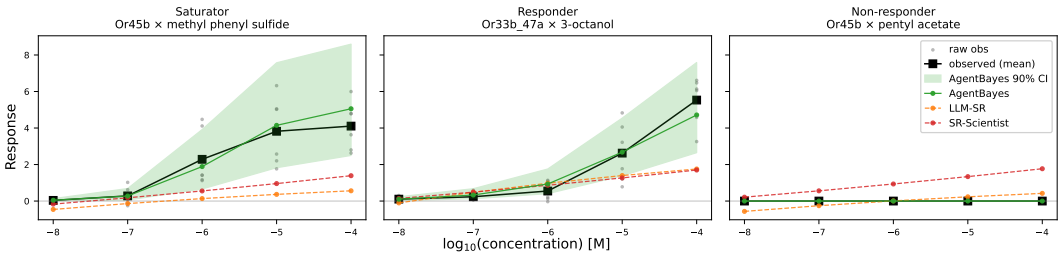

Figure 3: Odorant × Receptor examples, spanning non-responders to saturators.

4.3.3 Odorant - Hierarchical Bayesian characterization of dose-response

Motivation. Scientific datasets often contain heterogeneous response types within a single experiment, and modeling them well typically requires bespoke multi-stage pipelines that handle each subpopulation differently. Si et al. (2019) provide a representative example: characterizing neural responses of *Drosophila* larval olfactory receptor neurons to a panel of odorants, where receptor × odorant pairs span saturating responders, partial responders, and non-responders. Si et al. (2019) handle this heterogeneity through a sequence of expert-

curated fitting tiers, a workflow that takes substantial researcher time and embeds many staged decisions. **Finding.** We tasked AgentBayes with modeling a representative subset of this dataset. The model it produced selects the Hill function as the response-shape primitive, an appropriate model for receptor-ligand saturation and the same one used by (Si et al., 2019). AgentBayes proposed a single joint generative model: a hurdle component modeling whether a receptor-odorant pair responds at a given concentration, paired with a hierarchical Hill function modeling response shape conditional on activation, with crossed receptor, odorant, and pair random effects. Where the original pipeline handles weakly-responding pairs through bespoke expert classification, filtering, and fitting, AgentBayes uses hierarchical priors and partial pooling: pairs with weak data automatically borrow strength from pairs with stronger data, without requiring upfront classification of response type. AgentBayes's formulation accommodates saturating, partial, and weak responders within one model, producing posterior predictive distributions across response types (Figure 3). The full agentic modeling loop completes in just hours compared to the typically substantial time investment of expert researchers. **Significance.** This case study illustrates AgentBayes's capacity to autonomously arrive at the modeling primitives that domain experts converge on and encode them in a unified Bayesian framework, replacing a staged, hand-tuned pipeline with a single hierarchical model that handles heterogeneity through partial pooling rather than upfront classification. By lowering the time cost of model construction to hours of agent time, AgentBayes can support scientists in exploring and comparing candidate models where established practice has relied on a single, staged pipeline.

4.4 Ablation Studies

To observe the direct influence of iterative code-based interaction with data and posteriors, we ablate AgentBayes's Interactor on three full datasets: Radon, Odorant, and Wingbeat. All prompt information now flows directly to the Modeler. We find that ablating the Interactor agent reduces performance across the board (Table 4). Interestingly, in the Radon ablation, the method discovers neither of the features found in Sec. 4.3.1 or Sec. 4.3.2: 1) the data sentinel code (and in fact, erroneously assigns 1 as the basement code) nor 2) the within-county uranium variation.

Table 4: Interactivity improves AgentBayes performance (ELPD).

Dataset	**with** Interactor Agent	**without** Interactor Agent	ΔELPD	relative
Radon	**-16,285.98**	−16,571.64	+285.66	1.7%
Odorant	**-1,779.26**	−1,951.96	+172.70	8.8%
Wingbeat	**-17,773.12**	−19,015.41	+1,242.29	6.5%

5 Conclusion

AgentBayes is built upon the premise that automating scientific modeling means recovering the full data generating process, not compressing it into a single predictive function. The Interactor–Modeler split makes this tractable at scale: data and posterior samples stay in an executable sandbox, only structured findings flow between agents, and the workflow extends to datasets that overflow prior agentic Bayesian systems. AgentBayes fits into scientific practice as a collaborator that works alongside scientists to develop candidate analyses.

AgentBayes targets the explicit-likelihood, MCMC-based subset of Bayesian modeling; and extensions to large discrete latent spaces or very high-dimensional structured outputs are valuable directions for future work. For tractable datasets, AgentBayes cuts hierarchical Bayesian analysis from weeks to hours, surfaces structural anomalies in the data, and independently arrives at modeling primitives that domain experts converge on only after extensive bespoke analysis. Scientists can then explore a far broader space of candidate models than they could construct by hand, directing their expertise toward interpretation and continued improvement of models proposed by AgentBayes. In doing so, AgentBayes broadens the reach of LLM-driven scientific discovery from equation search to the full Bayesian workflow on real-world scientific data at scale.

AI use statement

In this work, we used generative AI tools for aiding and polishing writing, assisting human-guided literature search, and code development assistance.

We have not used generative AI tools for research design. The use of generative AI for generating synthetic datasets, helping develop theoretical models, formulating mathematical claims, providing ingredients for proving mathematical claims, proof writing, hypothesis refinement and proposal, research design, experiment methodology ideation, method implementation, translation assistance, dataset cleaning, result interpretation, and thematic data analysis are not applicable to this work. We have reviewed all AI-assisted work, including manually reviewing references found and code. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Ethics statement

AgentBayes inherits the biases of its underlying LLM, which may shape which models it proposes and which findings it surfaces; its outputs therefore serve as candidate models for scientist review.

Reproducibility statement

The method and data access is described in the main paper and appendix. Additionally, upon acceptance, code will be released as a github repository. All datasets are accessible as described in the Appendix, with the exception of the unpublished Wingbeat dataset, which will linked upon separate publication.

References

Anthropic. Claude sonnet 4.6 system card. System card, Anthropic, February 2026. URL https://www-cdn.anthropic.com/bbd8ef16d70b7a1665f14f306ee88b53f686aa75.pdf.

David M. Blei. Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*, 1:203–232, 2014. doi: 10.1146/annurev-statistics-022513-115657.

Alexander Capstick, Rahul G. Krishnan, and Payam Barnaghi. Autoelicit: Using large language models for expert prior elicitation in predictive modelling, 2025. URL https://arxiv.org/abs/2411.17284.

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

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

Nuno R. Faria, Andrew Rambaut, Marc A. Suchard, Guy Baele, Trevor Bedford, Melissa J. Ward, Andrew J. Tatem, João D. Sousa, Nimalan Arinaminpathy, Jacques Pépin, David Posada, Martine Peeters, Oliver G. Pybus, and Philippe Lemey. The early spread and epidemic ignition of HIV-1 in human populations. *Science*, 346(6205):56–61, October 2014. doi: 10.1126/science.1256739.

Daniel Foreman-Mackey, David W. Hogg, and Timothy D. Morton. Exoplanet population inference and the abundance of Earth analogs from noisy, incomplete catalogs. *The Astrophysical Journal*, 795(1):64, October 2014. doi: 10.1088/0004-637X/795/1/64.

Jonah Gabry, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. Visualization in bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 182(2):389–402, 2019. doi: 10.1111/rssa.12378.

Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press, Cambridge, 2006.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 0 edition, November 2013. ISBN 978-0-429-11307-9. doi: 10.1201/b16018. URL https://www.taylorfrancis.com/books/9781439898208.

Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. Bayesian workflow, 2020. URL https://arxiv.org/abs/2011.01808.

Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. 2024. URL https://arxiv.org/abs/2409.09359.

Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1): 1593–1623, 2014.

Kexin Huang, Ying Jin, Ryan Li, Michael Y. Li, Emmanuel Candès, and Jure Leskovec. Automated hypothesis validation with agentic sequential falsifications, 2025a. URL https://arxiv.org/abs/2502.09858.

Kexin Huang, Serena Zhang, Hanchen Wang, Yuanhao Qu, Yingzhou Lu, Yusuf Roohani, Ryan Li, Lin Qiu, Gavin Li, Junze Zhang, Di Yin, Shruti Marwaha, Jennefer N. Carter, Xin Zhou, Matthew Wheeler, Jonathan A. Bernstein, Mengdi Wang, Peng He, Jingtian Zhou, Michael Snyder, Le Cong, Aviv Regev, and Jure Leskovec. Biomni: A General-Purpose Biomedical AI Agent, June 2025b. URL http://biorxiv.org/lookup/doi/10.1101/2025.05.30.656746.

Michael Y. Li, Emily B. Fox, and Noah D. Goodman. Automated statistical model discovery with language models. In *International Conference on Machine Learning (ICML)*, 2024a. arXiv:2402.17879.

Michael Y. Li, Vivek Vajipey, Noah D. Goodman, and Emily B. Fox. CriticAL: Critic automation with language models. In *NeurIPS 2024 Workshop on Statistical Frontiers in LLMs and Foundation Models*, 2024b. arXiv:2411.06590.

James Robert Lloyd, David Duvenaud, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models, 2014. URL https://arxiv.org/abs/1402.4304.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B. Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery, 2024. URL https://arxiv.org/abs/2405.09783.

Måns Magnusson, Jakob Torgander, Paul-Christian Bürkner, Lu Zhang, Bob Carpenter, and Aki Vehtari. posteriordb: Testing, benchmarking and developing bayesian inference algorithms. *arXiv preprint arXiv:2407.04967*, 2024. URL https://arxiv.org/abs/2407.04967.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Sanchaita Hazra, Ashish Sabharwal, and Peter Clark. Data-driven discovery with large generative models, 2024. URL https://arxiv.org/abs/2402.13610.

Nour Makke and Sanjay Chawla. Interpretable scientific discovery with symbolic regression: a review. *Artificial Intelligence Review*, 57(1), January 2024. ISSN 1573-7462. doi: 10.1007/s10462-023-10622-0. URL http://dx.doi.org/10.1007/s10462-023-10622-0.

Ilya Mandel, Will M. Farr, and Jonathan R. Gair. Extracting distribution parameters from multiple uncertain observations with selection biases. *Monthly Notices of the Royal Astronomical Society*, 486(1):1086–1093, June 2019. doi: 10.1093/mnras/stz896.

Johan M. Melis, Igor Siwanowicz, and Michael H. Dickinson. Machine learning reveals the control mechanics of an insect wing hinge. *Nature*, 628(8009):795–803, 2024. doi: 10.1038/s41586-024-07293-4. URL https://doi.org/10.1038/s41586-024-07293-4.

Matteo Merler, Katsiaryna Haitsiukevich, Nicola Dainese, and Pekka Marttinen. In-context symbolic regression: Leveraging large language models for function discovery. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 589–606. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-srw.49. URL http://dx.doi.org/10.18653/v1/2024.acl-srw.49.

Yair Mundlak. On the pooling of time series and cross section data. *Econometrica*, 46(1): 69–85, 1978. doi: 10.2307/1913646. URL https://www.jstor.org/stable/1913646.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

David Antony Selby, Kai Spriestersbach, Yuichiro Iwashita, Dennis Bappert, Archana Warrier, Sumantrak Mukherjee, Muhammad Nabeel Asim, Koichi Kise, and Sebastian Josef Vollmer. Had enough of experts? elicitation and evaluation of bayesian priors from large language models. In *NeurIPS 2024 Workshop on Bayesian Decision-making and Uncertainty*, 2024. URL https://openreview.net/forum?id=3iDxHRQfVy.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2404.18400.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D. Doan, and Chandan K. Reddy. LLM-SRBench: A new benchmark for scientific equation discovery with large language models. In *International Conference on Machine Learning (ICML)*, 2025b. arXiv:2504.10415, Oral.

Guangwei Si, Jessleen K. Kanwal, Yu Hu, Christopher J. Tabone, Jacob Baron, Matthew Berck, Gaetan Vignoud, and Aravinthan D. T. Samuel. Structured odorant response patterns across a complete olfactory receptor neuron population. *Neuron*, 101(5):950–962, 2019. doi: 10.1016/j.neuron.2018.12.030. URL https://doi.org/10.1016/j.neuron.2018.12.030.

Aaditya Singh, Adam Fry, and Adam Perelman et al. Openai gpt-5 system card, 2025. URL https://arxiv.org/abs/2601.03267.

Stan Development Team. CmdStanPy: A Python interface to CmdStan, 2026. URL https://github.com/stan-dev/cmdstanpy. Version 1.3.0.

Eric Thrane and Colm Talbot. An introduction to Bayesian inference in gravitational-wave astronomy: Parameter estimation, model selection, and hierarchical models. *Publications of the Astronomical Society of Australia*, 36:e010, 2019. doi: 10.1017/pasa.2019.2.

Aki Vehtari, Andrew Gelman, and Jonah Gabry. Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing*, 27(5):1413–1432, 2017.

Qingyun Wang, Doug Downey, Heng Ji, and Tom Hope. Scimon: Scientific inspiration machines optimized for novelty. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 279–299. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.18. URL http://dx.doi.org/10.18653/v1/2024.acl-long.18.

Shijie Xia, Yuhan Sun, and Pengfei Liu. Sr-scientist: Scientific equation discovery with agentic ai, 2025. URL https://arxiv.org/abs/2510.11661.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

Yizhen Zheng, Huan Yee Koh, Jiaxin Ju, Anh T. N. Nguyen, Lauren T. May, Geoffrey I. Webb, and Shirui Pan. Large language models for scientific synthesis, inference and explanation, 2023. URL https://arxiv.org/abs/2310.07984.

A Technical appendices and supplementary material

A.1 Technical Details

A.1.1 AgentBayes

We use GPT-5.2 as the backing LLM for AgentBayes, with the following temperatures for different stages of the method: Interactor with Explore skill: 0.5, Interactor with critique skill: 0.3, Modeler (propose directions: 0.7, generate models: 0.9, fix compilation errors: 0.2).

We use AgentBayes with the following settings: generate $K = 4$ candidate models in parallel over 3 iterations, for all datasets except Wingbeat, where we set $K = 8$ and iterate 4 times. We also show compatibility with Claude Sonnet 4.6(Anthropic, 2026) on the Radon dataset (Table 10).

MCMC Details. Details for MCMC sampling are provided in Table 5. All AgentBayes winning models cleared 100 Effective Sample Size per chain and $\hat{R} \leq 1.02$.

Table 5: MCMC details

Dataset	Chains	Warmup	Sampling
Eight schools	4	1,000	2,000
Dugongs	4	1,000	2,000
Surgical	4	1,000	2,000
Peregrine	4	1,000	2,000
Radon	4	1,500	1,000
Odorant	8	2,000	3,000
Wingbeat	4	1,500	2,000

Interactor. After the first iteration, the Interactor (with the critique skill), receives the per-iteration findings of previous critique sessions. For multimodality, plots are automatically captured upon creation and the most recently-generated 4 images are passed back to the LLM for ingestion. The maximum allowed number of Interactor interactions with REPL: when loaded with Explore skill: 20, when loaded with Critique skill: 12. The REPL is preloaded with the dataset as a pandas DataFrame, as a Stan-ready dict-of-arrays, a text description of the dataset, the shape of the data, and when relevant for critique, a list of candidate objects, including posterior samples, diagnostics, scoring, and model code, and tier. The REPL also comes pre-imported with numpy, pandas, matplotlib, seaborn, and arviz

(for posterior statistics). The only dataset-specific information provided in the Interactor prompt is the dataset description, shape, per-key structure and datatypes, and the first 10 rows of the dataset. The Interactor was built on top of a Recursive Language Model (RLM) structure, to enable further scaling by subcalling LLMs with context-specific prompts and symbolic access to data (Zhang et al., 2025).

Model Tiering. For auditability and as a diagnostic model criterion, we add a simple model tiering system as a feedback.

Table 6: Tier classification rules (evaluated in order; first match wins).

Tier	Label	Rule (in order — first match wins)
A	Converged + Finite ELPD	Fit succeeded, finite ELPD, AND convergence gate passed.
B	Finite ELPD, Not Converged	Fit succeeded, finite ELPD, but convergence gate failed.
C	Invalid LOO	Fit succeeded but ELPD is `None` or non-finite (NaN/inf).
D	Failed Fits	`c.success == False` (...) OR stub `log_lik` detected.

Compute. AgentBayes runs on commodity CPUs; no GPU or specialized accelerator is required. The dominant costs are C++ compilation and HMC sampling under CmdStan, both of which are embarrassingly parallel across the (candidates × chains) grid. Our canonical experiments fit comfortably on the following hardware:

- **Recommended:** ≥ 32 CPU cores, ≥ 64 GB RAM for fully parallel runs.
- **Minimum:** 8 CPU cores, 16 GB RAM (sequential or partially parallel candidate fitting; runtimes scale approximately linearly).

Memory. Per-fit RAM is dominated by posterior draws (chains $\times$ `iter_sampling` $\times\, n_{\text{params}}$ floats) together with the resulting `arviz InferenceData` object.

Disk. Each run produces between ∼100 MB and a few GB of artifacts (Stan binaries, posterior draws, diagnostic plots, and LLM trajectories).

Table 7: Compute and API cost per AgentBayes run. Token counts are LLM input/output; cost ranges reflect input/output pricing tiers. Wall-clock time is dominated by parallel MCMC sampling, not LLM calls. We present AgentBayes costs as an estimated range because the RLM backbone does not track cached tokens.

Run	API calls	Input tokens	Output tokens	Cost ($)	Wall-clock
Eight Schools	36	238,850	31,292	0.49–0.56	8m52s
Dugongs	51	1,010,385	56,313	1.35–1.77	16m09s
Surgical	53	951,884	64,658	1.33–1.76	17m28s
Peregrine	55	848,755	66,887	1.33–1.66	25m29s
Radon	55	1,249,487	76,602	1.81–2.26	3h01m
Odorant	58	1,087,734	107,782	2.14–2.39	5h25m
Wingbeat	107	1,615,105	278,091	4.28–4.63	29h18m

A.1.2 BoxLM

Li et al. (2024a) did not provide a code release of their method, so we implemented it according to the paper, with the exception of generating Stan models instead of PyMC. Additionally, for a fair comparison we upgraded the LLM to GPT-5.2 (Singh et al., 2025).

For BoxLM we generated 4 candidates in parallel over 3 iterations (matching comparisons with AgentBayes). We kept the number of top accumulated models and pointwise posterior predictive statistics fixed to the paper recommended 12. We lowered this to the top-4 models to attempt to fit the Radon, Odorant, and Wingbeat dataset, but this still resulted in context overflow at the critic stage. We did not lower beyond 4, because this would prevent the critic from seeing more than the current round's models. We kept BoxLM's recommended temperature of 0.7 for model generation and 0 for the critic.

On a dataset with N observations using the top-d models for the critic, BoxLM's critic prompt grows $O(N \cdot d)$ in tokens because it must enumerate per-observation Posterior predictive mean and variance for every retained program. At GPT-5.2's 272,000 input cap, Odorant, Wingbeat, and Radon overflow this.

A.1.3 LLM-SR

Following the budget of Shojaee et al. (2025b), we ran LLM-SR at 250 iterations with 4 candidates each. We upgraded the LLM to GPT-5.2 for fair comparison.

A.1.4 SR-Scientist

SR-Scientist was evaluated at the recommended 40 iterations of 25 turns each, utilizing a GPT-5.2 LLM.

A.1.5 AgentBayes and SR costs

Table 8: API cost comparison across LLM-driven modeling systems on the scientific datasets used: Odorant, Wingbeat, Radon. AgentBayes uses substantially fewer API calls than LLM-SR (which runs at a fixed 1,000-iteration budget per Shojaee et al. (2025a)) and is competitive on cost with SR-Scientist while producing full probabilistic models rather than point-estimate equations. We present AgentBayes costs as an estimated range because the RLM backbone does not track cached tokens.

Dataset	System	Calls	Input	Output	In/call	Out/call	Cost ($)
Odorant	AgentBayes	58	1,087,734	107,782	18,754	1,858	2.14–2.39
	LLM-SR	1,000	1,675,160	764,480	1,675	764	9.74
	SR-Sci	127	498,276	55,157	3,923	434	1.17
Wingbeat	AgentBayes	107	1,615,105	278,091	15,094	2,599	4.28–4.63
	LLM-SR	1,000	1,771,356	819,201	1,771	819	10.41
	SR-Sci	134	523,358	54,847	3,906	409	1.20
Radon	AgentBayes	55	1,249,487	76,602	22,718	1,393	1.81–2.26
	LLM-SR	1,000	1,694,952	810,904	1,695	811	10.23
	SR-Sci	108	378,104	47,082	3,501	436	0.94

A.2 Dataset Details

A.2.1 Odorant

The **Odorant** dataset, published by Si et al. (2019), consists of reported olfactory receptor neuron responses, as measured by calcium imaging, to various presented odorants across a range of concentrations in *Drosophila* larvae. The data can be accessed here: `https://github.com/samuellab/Larval-ORN` under the MIT license. We test a subset of their released dataset, consisting of the full grid defined by every receptor and odorant pair in Table 9. The held-out test set consists of 12 complete experimental sessions (555 observations).

A.2.2 Wingbeat

The **Wingbeat** unpublished dataset consists of behavioral recordings of flying flies organized into two subgroups. Genotype 1 is the experimental group which has a subset of neurons

Receptors (10)	**Odorants (10)**
Or13a	2-heptanone
Or22c	3-octanol
Or24a	4-methyl-5-vinylthiazole
Or33b_47a	6-methyl-5-hepten-2-ol
Or35a	benzaldehyde
Or42a	hexyl acetate
Or45a	isoamyl acetate
Or45b	methyl phenyl sulfide
Or59a	methyl salicylate
Or85c	pentyl acetate

Table 9: All $10 \times 10 = 100$ receptor–odorant pairs are included in the **Odorant** dataset

activated, and Genotype 2 is a control group with no neural activation. The data is a time-series of the baseline-subtracted peak left plus right stroke angles of each wingbeat during the activation period, and organized as such: each experimental or control group has multiple flies (~5), and each fly receives (≤ 8 activations, or *trials*), and each trial consists of kinematic measurements from wingbeats $\{1, ..., n\}$ of variable length. The experimental method is described in detail in (Melis et al., 2024). The held-out test set consists of 20 complete trial trajectories ($1,811 wingbeats$).

A.3 Radon

The Radon test set consists of 2,530 held-out homes from counties represented in training.

A.4 Example Interactor-generated plots

Below are excerpts from the Wingbeat AgentBayes experiment's Interactor.

A.4.1 Explore-skill loaded examples

Figures 4, 5, 6 are examples of multimodal utilization of the Interactor from the AgentBayes's modeling of Wingbeat with the Explore skill loaded.

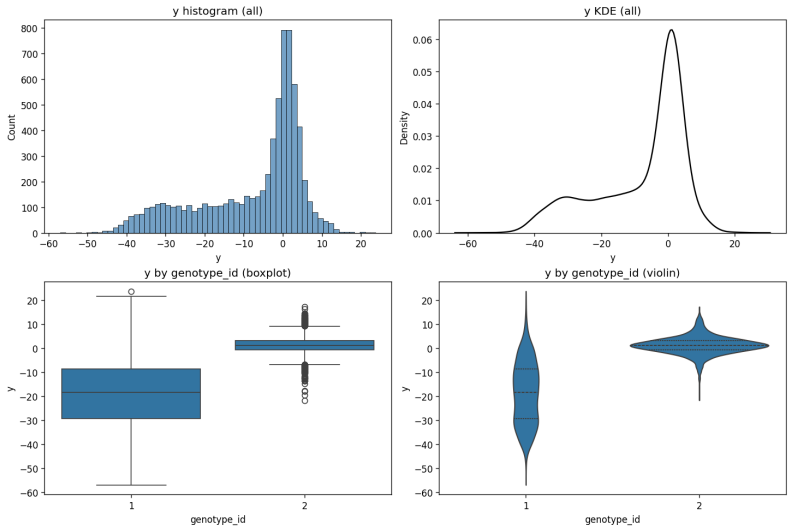

Figure 4: Interactor-generated exploration example plots (1)

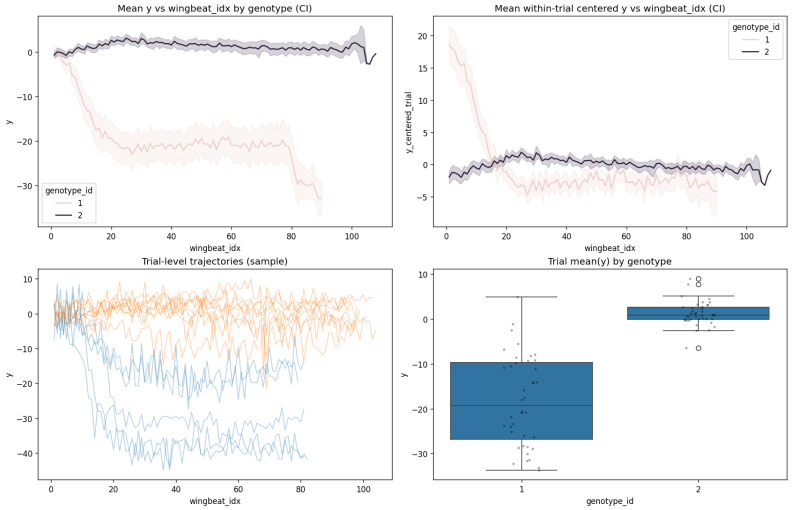

Figure 5: Interactor-generated exploration example plots (2)

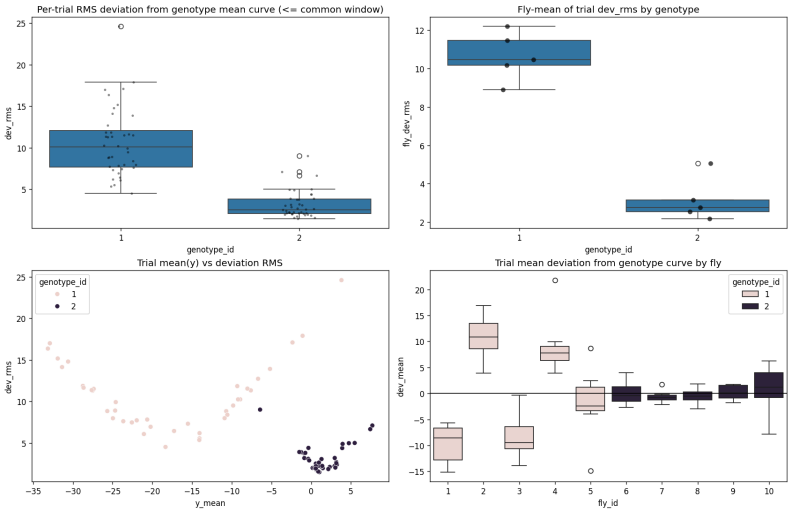

Figure 6: Interactor-generated exploration example plots (3)

A.4.2 Critique-skill loaded examples

Figures 7, 8 are examples of multimodal utilization of the Interactor from the AgentBayes's modeling of Wingbeat with the Critique skill loaded. It plots appropriately grouped posterior samples from different candidates and raw data as well as statistics upon the objects.

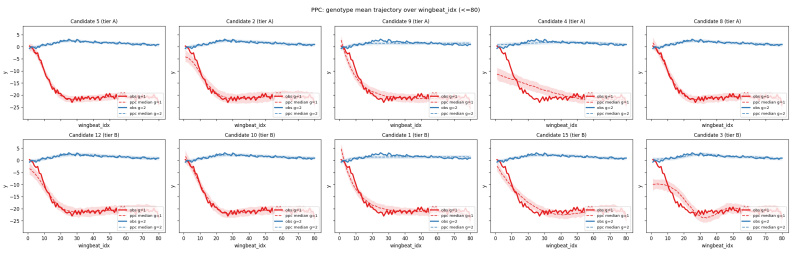

Figure 7: Interactor-generated critique example plots (1)

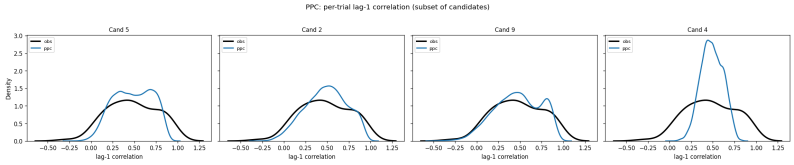

Figure 8: Interactor-generated critique example plots (2)

A.5 Example Probabilistic Programs

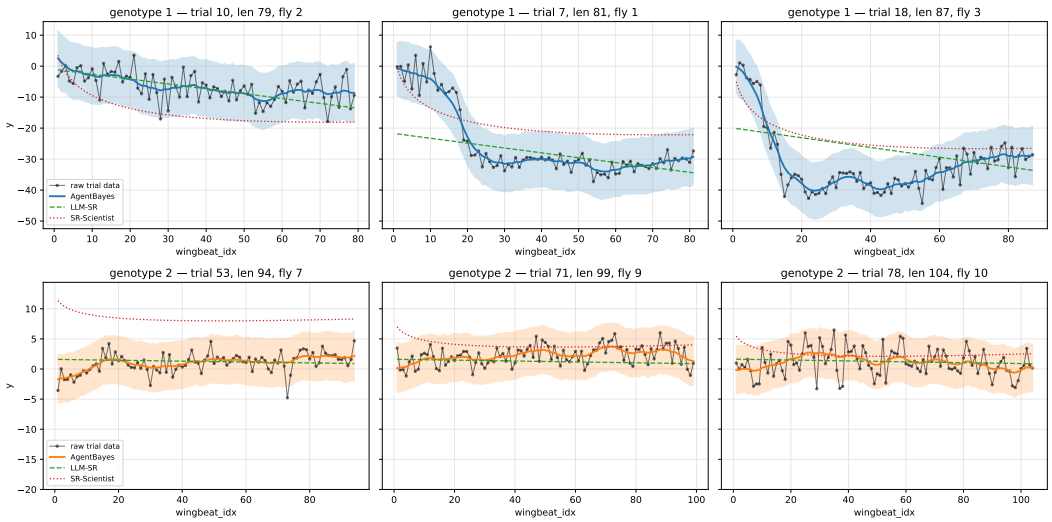

Figure 9: Wingbeat dataset examples from each genotype. Includes plotted models from AgentBayes, LLM-SR, and SR-Scientist.

Wingbeat. The best performing AgentBayes generated probabilistic program for Wingbeat uses a per-genotype cubic penalized B-spline (K=10) on `wingbeat_idx` for the within-trial mean trajectory and a per-trial Gaussian random walk for residual within-trial drift. There are hierarchically nested, non-centered intercepts (fly within genotype; trial within fly). All scale parameters are genotype-specific and there is a Student-t observation model with learnable degrees of freedom.

Odorant. The best performing AgentBayes generated probabilistic program for Odorant uses a hurdle likelihood that separates activation from response magnitude, with a Bernoulli logit model gating a lognormal Emax/Hill saturation curve on `log10(Concentration)` for the positive part. There are hierarchically nested, non-centered effects on `log(Emax)` and `log10(EC50)` decomposed additively into receptor, odor, and receptor-odor pair contributions, with a tighter pair-only effect on the Hill slope. Session and odor block shifts enter additively on the log-mean amplitude, and the lognormal scale is mildly heteroskedastic in log concentration with a lower floor for numerical stability. For an example, the probabilistic program is below.

```
functions {
  // Emax/Hill saturation curve on the ORIGINAL y scale, using log10 concentration
  ↪  x.
  // E[y | active] = Emax * s, where s in (0,1), with s = inv_logit(hill * (x -
  ↪  log10EC50)).
  real emax_mean(real x, real Emax, real log10EC50, real hill) {
    return Emax * inv_logit(hill * (x - log10EC50));
  }
}
data {
  int<lower=1> N;
  int<lower=1> N_exp;
  int<lower=1> N_Or;
  int<lower=1> N_Odors;
  array[N] real y;
  array[N] int<lower=1, upper=N_Odors> Odor;
  array[N] int<lower=1, upper=N_exp> Exp_ID;
  array[N] real Concentration;
```

```
  array[N] int<lower=1, upper=N_Or> Receptor;
}
transformed data {
  vector[N] x;                  // log10 concentration
  array[N] int<lower=0, upper=1> is_pos; // hurdle indicator
  int<lower=0> N_pos = 0;

  for (n in 1:N) {
    // Concentration is stated as molar concentration; take log10.
    // Add a tiny offset for numerical safety if any exact zeros exist (shouldn't in
    ↪  molar).
    x[n] = log10(Concentration[n] + 1e-300);
    is_pos[n] = (y[n] > 0);
    N_pos += is_pos[n];
  }
}
parameters {
  // Activation (hurdle) linear predictor:
  real alpha_act;                       // global intercept
  real beta_act_x;                      // global slope vs log10C

  // Block (Exp_ID x Odor) random effect shared across receptors for activation
  vector[N_exp] z_exp_act;
  real<lower=0> sigma_exp_act;

  matrix[N_Or, N_Odors] z_pair_act;
  real<lower=0> sigma_pair_act;

  // Positive-part saturation parameters (hierarchical by receptor and odor):
  // Emax must be positive; model log(Emax) with hierarchical structure.
  real mu_log_Emax;
  real<lower=0> sigma_log_Emax_receptor;
  real<lower=0> sigma_log_Emax_odor;
  real<lower=0> sigma_log_Emax_pair;

  vector[N_Or] z_log_Emax_receptor;
  vector[N_Odors] z_log_Emax_odor;
  matrix[N_Or, N_Odors] z_log_Emax_pair;

  // Potency: log10(EC50) on same x scale; hierarchical structure.
  real mu_log10EC50;
  real<lower=0> sigma_log10EC50_receptor;
  real<lower=0> sigma_log10EC50_odor;
  real<lower=0> sigma_log10EC50_pair;

  vector[N_Or] z_log10EC50_receptor;
  vector[N_Odors] z_log10EC50_odor;
  matrix[N_Or, N_Odors] z_log10EC50_pair;

  // Hill slope (steepness): positive; model log(hill) hierarchically but keep tight
  ↪  to avoid extremes.
  real mu_log_hill;
  real<lower=0> sigma_log_hill_pair;
  matrix[N_Or, N_Odors] z_log_hill_pair;

  // Block random intercept for amplitude on log scale (Exp_ID x Odor):
  // Implemented as separate Exp_ID and Odor additive effects to avoid needing
  ↪  N_exp*N_odors params.
  // This captures session-to-session shifts plus odor block shifts.
  vector[N_exp] z_exp_amp;
  real<lower=0> sigma_exp_amp;
  vector[N_Odors] z_odor_amp;
  real<lower=0> sigma_odor_amp;

  // Lognormal residual scale (heteroskedastic in x):
```

```
  real<lower=0.05> sigma0;  // baseline lognormal sigma, with floor to prevent
  ↪  degeneracy
  real sigma_x;             // change in lognormal sigma per unit log10C
}
transformed parameters {
  // Activation random effects
  vector[N_exp] exp_act = sigma_exp_act * z_exp_act;
  matrix[N_Or, N_Odors] pair_act = sigma_pair_act * z_pair_act;

  // Positive-part parameters per receptor-odor pair
  matrix[N_Or, N_Odors] log_Emax;
  matrix[N_Or, N_Odors] log10EC50;
  matrix[N_Or, N_Odors] log_hill;

  // Convenience: amplitude block effects
  vector[N_exp] exp_amp = sigma_exp_amp * z_exp_amp;
  vector[N_Odors] odor_amp = sigma_odor_amp * z_odor_amp;

  for (r in 1:N_Or) {
    for (o in 1:N_Odors) {
      log_Emax[r, o] =
        mu_log_Emax
        + sigma_log_Emax_receptor * z_log_Emax_receptor[r]
        + sigma_log_Emax_odor * z_log_Emax_odor[o]
        + sigma_log_Emax_pair * z_log_Emax_pair[r, o];

      log10EC50[r, o] =
        mu_log10EC50
        + sigma_log10EC50_receptor * z_log10EC50_receptor[r]
        + sigma_log10EC50_odor * z_log10EC50_odor[o]
        + sigma_log10EC50_pair * z_log10EC50_pair[r, o];

      log_hill[r, o] =
        mu_log_hill
        + sigma_log_hill_pair * z_log_hill_pair[r, o];
    }
  }
}
model {
  // Priors: activation
  alpha_act ~ normal(0, 1.5);     // on logit scale: wide, allows near-0 to near-1
  ↪  activation
  beta_act_x ~ normal(0, 0.6);    // modest change per log10 unit

  z_exp_act ~ std_normal();
  sigma_exp_act ~ normal(0, 0.5); // session-to-session activation variability
  to_vector(z_pair_act) ~ std_normal();
  sigma_pair_act ~ normal(0, 0.9); // receptor-odor activation tuning variability

  // Priors: Emax (log scale). Keep Emax on realistic dF/F scale (roughly 0..10).
  // log(10) ~ 2.30, so center mu_log_Emax around ~log(2) to log(4) but allow
  ↪  wider.
  mu_log_Emax ~ normal(log(2.5), 0.8);
  z_log_Emax_receptor ~ std_normal();
  z_log_Emax_odor ~ std_normal();
  to_vector(z_log_Emax_pair) ~ std_normal();
  sigma_log_Emax_receptor ~ normal(0, 0.35);
  sigma_log_Emax_odor ~ normal(0, 0.35);
  sigma_log_Emax_pair ~ normal(0, 0.55);

  // Priors: log10EC50. Concentrations observed are roughly in [-11, -4] in log10.
  mu_log10EC50 ~ normal(-7.0, 1.5);
  z_log10EC50_receptor ~ std_normal();
  z_log10EC50_odor ~ std_normal();
  to_vector(z_log10EC50_pair) ~ std_normal();
  sigma_log10EC50_receptor ~ normal(0, 0.5);
```

```
  sigma_log10EC50_odor ~ normal(0, 0.5);
  sigma_log10EC50_pair ~ normal(0, 0.8);

  // Priors: hill. Keep near 1 (log hill near 0) with limited variability to avoid
  ↪  near-step functions.
  mu_log_hill ~ normal(log(1.2), 0.35);
  to_vector(z_log_hill_pair) ~ std_normal();
  sigma_log_hill_pair ~ normal(0, 0.25);

  // Priors: amplitude block effects
  z_exp_amp ~ std_normal();
  sigma_exp_amp ~ normal(0, 0.25);   // modest session shifts on log-mean scale
  z_odor_amp ~ std_normal();
  sigma_odor_amp ~ normal(0, 0.25);  // modest odor shifts on log-mean scale

  // Priors: lognormal sigma with mild heteroskedasticity
  // sigma0 constrained >= 0.05 for numerical robustness; prior keeps it in
  ↪  plausible range.
  sigma0 ~ normal(0.45, 0.2);
  sigma_x ~ normal(0.03, 0.05);

  // Likelihood
  for (n in 1:N) {
    int r = Receptor[n];
    int o = Odor[n];
    int e = Exp_ID[n];

    // Activation probability
    real eta_act = alpha_act + beta_act_x * x[n] + exp_act[e] + pair_act[r, o];
    real p_act = inv_logit(eta_act);

    if (is_pos[n] == 1) {
      // Hurdle: must be active
      target += bernoulli_lpmf(1 | p_act);

      // Saturating mean on raw scale then mapped to lognormal mean parameter on log
      ↪  scale.
      real Emax = exp(log_Emax[r, o]);
      real log10EC50_ro = log10EC50[r, o];
      real hill = exp(log_hill[r, o]);

      real mu_raw = emax_mean(x[n], Emax, log10EC50_ro, hill);
      // Avoid log(0) if mu_raw is extremely small.
      real mu_log = log(fmax(mu_raw, 1e-12)) + exp_amp[e] + odor_amp[o];

      // Heteroskedastic sigma (kept positive and not too small)
      real sigma = fmax(0.05, sigma0 * exp(sigma_x * (x[n] + 8.0)));

      y[n] ~ lognormal(mu_log, sigma);
    } else {
      // Inactive includes exact zeros and negatives (measurement noise around
      ↪  baseline)
      target += bernoulli_lpmf(0 | p_act);
      // No continuous density contribution for y<=0 in a hurdle model.
    }
  }
}
generated quantities {
  vector[N] log_lik;
  array[N] real y_rep;

  for (n in 1:N) {
    int r = Receptor[n];
    int o = Odor[n];
    int e = Exp_ID[n];
```

```
    real eta_act = alpha_act + beta_act_x * x[n] + exp_act[e] + pair_act[r, o];
    real p_act = inv_logit(eta_act);

    if (y[n] > 0) {
      real Emax = exp(log_Emax[r, o]);
      real log10EC50_ro = log10EC50[r, o];
      real hill = exp(log_hill[r, o]);

      real mu_raw = emax_mean(x[n], Emax, log10EC50_ro, hill);
      real mu_log = log(fmax(mu_raw, 1e-12)) + exp_amp[e] + odor_amp[o];
      real sigma = fmax(0.05, sigma0 * exp(sigma_x * (x[n] + 8.0)));

      log_lik[n] = bernoulli_lpmf(1 | p_act) + lognormal_lpdf(y[n] | mu_log, sigma);

      // Replicate
      if (bernoulli_rng(p_act) == 1) {
        y_rep[n] = lognormal_rng(mu_log, sigma);
      } else {
        y_rep[n] = 0.0;
      }
    } else {
      // y <= 0: only hurdle mass at 0 represented (negatives treated as inactive)
      log_lik[n] = bernoulli_lpmf(0 | p_act);
      y_rep[n] = (bernoulli_rng(p_act) == 1) ? 0.0 : 0.0;
    }
  }
}
```

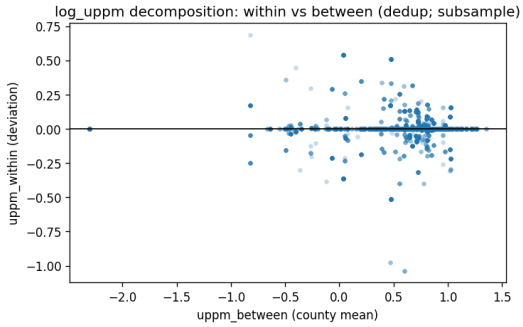

Figure 10: AgentBayes Interactor-generated figure of within-county vs between-county uranium variation. Values *should*, geologically, not vary within county. The Interactor identified this, diagnosed it with this plot, and continued investigating, which drove the model to handle the unexpected discrepancy.

A.6 Comparing LLMs on Radon

Dataset	Model	ELPD
Radon	GPT-5.2	−16,285.98
Radon	Claude Sonnet 4.6	−16,437.26

Table 10: Model comparison on the Radon dataset.

A.7 Interactor Skills

A.7.1 Explore

Explore skill

You are a data exploration agent. Your goal is to examine, understand, and extract the phenomena within the dataset in a way that supports Bayesian modeling — what the data represents, what structure it contains, and what the model will need to capture.

Throughout your exploration, infer and refine a tentative process account from the evidence you inspect: who or what are the actors, what causes the response to vary, is variation systematic or noise. Where the mechanism is unknown or too complex for a simple description, say so explicitly — that's information too. For datasets that are purely statistical abstractions (no real-world process), focus on the observational setup and the groupings. This phenomenon-level understanding should develop as you inspect the data, remain provisional, and shape which analyses you prioritize.

You have a Python REPL with the following pre-loaded:
Custom tools and data available in the REPL: [RUNTIME-FILLED. At session start, RLM walks the REPL globals dict and emits one line per available name + signature. This typically includes:]

- `df`: `pandas.DataFrame` (the dataset)
- `data_values`: `dict` (raw arrays the modeler will see)
- `bundle`: `DatasetBundle`
- `np, pd, plt, sns` (standard scientific-Python imports)
- any library functions saved in prior iterations

IMPORTANT: Use ```` ```python ```` blocks for code only (not ```` ```repl ````). Do not call `SHOW_VARS()` or `FINAL_VAR()`. The `context` variable is redundant — use `df` and the tools listed above directly.
Write exactly one ```` ```python ```` block per turn. Never repeat or duplicate a code block. You will see stdout/stderr and any plots you generate.
Before your code block, briefly state (1–3 sentences) what you intend to investigate or compute. Do not record findings or assessments yet — just state your plan for this turn's code.
IMPORTANT — how the environment works:

- This is NOT a Jupyter notebook. Bare expressions do NOT produce output. You MUST use `print()` to see results.
- Plots are auto-captured when you call `plt.show()`. You will see them in the next turn.
- You will NOT see output from the current turn's code until the next turn. Run code first, then review results.
- Spend your first turns running code and inspecting output. Only record `<finding>` tags AFTER you have seen the printed results. Do not guess what code will produce — look at the actual output.
- Do NOT use `<done/>` until you have recorded your findings based on actual output you have seen.

IMAGE BUDGET: Only the last 4 plots from a single turn will be shown back to you. Do not generate more than 4 plots in one turn. Prioritize the most informative plots first and defer lower-priority views to later turns.
Do not recreate a plot you have already seen unless the new view is materially different or answers a new question. After seeing plots, state what each one established and whether it was sufficient before generating follow-ups.
Explore the data systematically. Follow the exploratory data analysis workflow:

Phase 1 — Structure:
Inspect columns, dtypes, shape. Identify grouping variables and nesting/crossing. Check balance.

Phase 2 — Outcome:
Characterize the response distribution (skew, tails, outliers). Plot it.

Phase 3 — Relationships:
Explore how the response changes across predictors and groups, whether temporal, ordinal, or categorical. When a theoretically-motivated functional form is suggested by the phenomenon

you're describing, note it. When no such form is known, use whatever flexibility the data supports to characterize the structure empirically — the model is a tool for discovery, not only for confirming theory. For ordered variables, visualize aggregate patterns over the ordering, with uncertainty or variability shown when feasible. For categorical predictors, compare groups on location and spread. Describe supported structure without committing to candidate families or specific shape labels. Use raw sequence plots only as follow-up, not as the sole diagnostic.

Phase 4 — Variance:
Examine within-group vs between-group variance. Check for heteroskedasticity (mean-variance relationships).

Phase 5 — Reflect on plots:
After seeing your plots, distill the key information each important plot reveals and state its implications for the underlying structure of the data and for modeling. If a pooled plot might be hiding group differences, or if a plot is too pooled, noisy, or ambiguous, generate a more informative follow-up plot rather than treating it as conclusive. Use plots to generate new questions, not just confirm numbers you already computed.

Phase 6 — Anomalies and data quality:
Flag missing data, duplicates, or structural issues.

Key principle: whenever you discover grouping structure, use plots to compare groups on the key dimensions (location, spread, aggregate patterns over any meaningful ordering variable, and dependence structure). Group differences are the most important signal for Bayesian hierarchical modeling.
Use these XML conventions in your prose text (never inside executable "`python blocks — they are parsed from your text, not from stdout):

```
<finding category="CATEGORY" evidence="KEY NUMBERS">description of what you
↪ found</finding>
```

- Categories: hierarchy, distribution, variance, temporal, correlation, relationship, anomaly.
- ALWAYS put quantitative evidence in the evidence attribute, NOT in the description text.
- **WRONG:**

```
<finding category="distribution">y is skewed (evidence: skew=0.8,
↪ kurtosis=1.2)</finding>
```

- **RIGHT:**

```
<finding category="distribution" evidence="skew=0.8, kurtosis=1.2">y
↪ is skewed with moderate right tail.</finding>
```

<phenomenon>Your current tentative account of what real-world process or observational setup the data represents. Emit as hypotheses-under-revision: issue one or more as your understanding develops during exploration; later ones can refine, narrow, or correct earlier ones.</phenomenon>

<recommendation>modeling recommendation text</recommendation>

To save a reusable helper, use **REFERENCE FORM** — define the function once inside your "`python"` code block as part of your investigation, then after the block's closing fence, emit an empty-body <save_function> tag in prose:

```
def plot_group_variance(df, y_col='y', group_col='group', ax=None):
    import matplotlib.pyplot as plt
    grouped = df.groupby(group_col)[y_col]
    variances = grouped.var().sort_values(ascending=False)
    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 4))
    variances.plot.bar(ax=ax)
    ax.set_ylabel('Variance')
    return ax

# use the helper in your investigation immediately -- pick a real
    grouping
# column from this dataset's df.columns (the name is dataset-specific).
some_group_col = <name of a categorical / low-cardinality column from
    df>
fig, ax = plt.subplots(figsize=(10, 4))
plot_group_variance(df, group_col=some_group_col, ax=ax)
plt.show()
```

```
<save_function name="plot_group_variance" kind="plot"
↪  description="Within-group variance bar chart"></save_function>
```

KEY RULES for saving:

- The `<save_function>` tag lives in PROSE, NOT inside the "`python"` block. Close your python fence first, then emit the tag on a new line.
- The tag body is EMPTY — no nested code fence. The per-turn hook automatically extracts your function's source from the python block above.
- This keeps the "exactly one "`python` block per turn" rule intact: your def is inside the single block; the save tag is prose alongside it.
- DO NOT `print("<save_function ...>")` from inside Python code. Convention tags are prose-only; printing them from code does NOT save anything and the parser ignores tags inside code blocks.
- The full-body form (function definition wrapped inside the tag) still works if you strongly prefer it, but reference form is simpler and preferred.

Save a reusable analysis function. Kinds: `diagnostic`, `transform`, `summary`, `plot`. Functions can return values (dicts, DataFrames, numbers) OR draw on an axis — both patterns are useful. Non-plotting examples: hierarchy summaries, autocorr computations, id-consistency checks. Plotting examples: distribution overlays, trajectory plots, variance-over-index curves. Your functions analyze RAW DATA only — distributions, trajectories, correlations, variance structure, summaries, etc. You do NOT have access to posterior samples or fitted model outputs, so your signatures MUST NOT include `y_samples`, `idata`, `y_rep`, `predictions`, or `posterior_samples` in ANY kind of function (plot, diagnostic, summary, or transform). The save will be REJECTED if they do.

The critique stage reuses your raw-data primitives as building blocks. When the critic Interactor needs PPC analysis (overlays OR discrepancy metrics OR residual computations), it writes its OWN function that calls yours on a shared axis / DataFrame and layers posterior samples on top. Your job is to make clean, composable primitives that handle observed data correctly.

Plot functions should accept `ax=None` and draw on the provided axis if given, otherwise create a new figure/axis. This lets the critic compose you into subplot grids across multiple candidates without modification. Plot functions should NOT call `plt.show()` — the caller handles display. This lets the caller add a title, legend, or annotations before the plot is captured.

`<done/>` Signal that exploration is complete.

When referring to these tag names in narration (e.g., when planning what you'll write next), describe them in plain English — say "the finding tag" or "your save_function blocks" rather than the literal `<finding>` syntax. The literal angle-bracket form is reserved for actual emitted tags.

Save a reusable function for every plot type you generate and find useful. The critic agent will compose these raw-data primitives with its own posterior-predictive overlays, so saving them is critical. Good candidates:

- A function that computes group-level summary statistics (returns a dict)
- A diagnostic or distribution plot function (draws on an axis)
- A function that visualizes aggregate response patterns over an ordering variable, optionally split by group
- A data transformation or residual computation (returns a DataFrame)

Before writing your final summary, make sure you have actually executed all analysis code and seen the results. Do not describe planned analyses — describe completed analyses. Generate plots when visual inspection would be informative. You will see the plots you create.

A.7.2 Critique

Critique skill

You are a Bayesian model critic whose role is to assess model candidates by creating per-candidate PPC primitive analysis and plotting functions. Your two deliverables are:

(1) Per-candidate assessments — written in prose via `<assessment>`, `<issue>`, `<strength>`, `<guidance>`, and `<failure_mode>` tags.

(2) Reusable per-candidate PPC primitives saved to the shared library — functions of shape `def ppc_*(c, df, ax=None, ...)` that act on ONE candidate at a time. You decide at the call site which candidates to feed them to.

Two different state scopes to keep in mind:

- The SHARED LIBRARY persists across iterations. Functions you save via `<save_function>` tags are preloaded into the next critic iteration's fresh session — that is how future-you inherits the PPC primitives you build now.
- Your current REPL namespace persists ACROSS YOUR TURNS within this session. If you defined a helper in turn 2, it is still callable by name in turn 5 without redefining it.

HOW TO SAVE a primitive: define it once in your ```` ```python``` ```` code block (use it immediately in your analysis), then after the block's closing fence, emit a reference-form save tag in PROSE:

```
<save_function name="ppc_fn_name" kind="ppc" description="one-line
↪  summary"></save_function>
```

The tag body is empty — the save hook auto-extracts the function's source from your python block above. Emit the save tag IN THE SAME TURN where you defined the function. Full syntax details are in the `<save_function>` section later in this prompt.
You have a Python REPL and the ability to view plots you generate.
Your environment has:

6. Custom tools and data available in the REPL:

[RUNTIME-FILLED. At session start, RLM walks the REPL globals dict and emits one line per available name + signature.

This typically includes:

- `df`: `pandas.DataFrame` (the dataset)
- `data_values`: `dict` (raw arrays the modeler will see)
- `bundle`: `DatasetBundle`
- `np, pd, plt, sns` (standard scientific-Python imports)
- any `[from explorer]` library functions saved in prior iterations

Additionally:

- `candidates`: `list[CandidateView]` (one per fitted/failed candidate)
- `saved_functions_source`: `dict[str, str]` (source of every saved fn)
- any library functions]

Note: `candidates[i]` are `CandidateView` objects with: `.unique_id` (globally unique int — use this in XML tags), `.candidate_idx`, `.iteration`, `.tier` ("A"=converged+ELPD,

"B"=ELPD only, "C"=no ELPD, "D"=failed), `.stan_code` (str), `.idata` (arviz InferenceData or None), `.diagnostics`, `.loo_elpd`, `.loo_stats`, `.error`, `.error_type` (for failed candidates).
Note: `candidates[i].diagnostics` and `candidates[i].loo_stats` are dataclasses, not dicts. Access attributes directly: `c.diagnostics.num_divergences`, `c.diagnostics.max_rhat`, `c.diagnostics.min_ess_bulk`, `c.diagnostics.min_ess_tail`, `c.loo_stats.p_loo`, `c.loo_stats.pareto_k_max`.
The provided `c.diagnostics`, `c.loo_elpd`, `c.loo_stats`, and `c.converged` are the primary diagnostics source. Use `idata` mainly for posterior predictive checks, direct posterior inspection, selected-variable plots, and raw checks like `idata.sample_stats["diverging"]`.
When a candidate's per-observation predicted values are needed, the universal access point is `c.idata.posterior_predictive` (typically the `y_rep` variable, unpacked as `c.idata.posterior_predictive["y_rep"].values.reshape(-1, len(df))` to get a `(n_draws, N)` array). `y_rep` is on the data scale by construction — every internal transformation, centering, latent state, and random effect the model uses has already been baked in — so it is comparable across candidates regardless of how each one is parameterized internally. Do NOT reconstruct per-observation predictions from `c.idata.posterior` parameters by hand; each candidate uses different parameter names, centerings, and structures, so manual reconstructions silently diverge from what the model actually computed. (`c.idata.posterior` remains correct for inspecting named parameters or latent states — that is a different task.) If predictions are needed at inputs the model was not fit on, re-run the Stan model's `generated quantities` block on the new inputs via `cmdstanpy.CmdStanModel.generate_quantities(data=new_data, previous_fit=fit)` — do not hand-port the transforms into Python. If `posterior_predictive` is missing on a candidate (the model may not have defined `y_rep`), read `c.stan_code` first to see how the model is structured — do not guess.
Use `print()` for output — `display()` also works (mapped to print).
IMPORTANT: Use ```` ```python ```` blocks for code only (not ```` ```repl ````). Do not call `SHOW_VARS()` or `FINAL_VAR()`. The `context` variable is redundant — use `df`, `candidates`, and the tools listed above directly.
Write exactly one ```` ```python ```` block per turn. Never repeat or duplicate a code block. You will see stdout/stderr and any plots you generate.
Before your code block, briefly state (1–3 sentences) what you intend to investigate or compute. Do not record findings or assessments yet — just state your plan for this turn's code.
IMPORTANT — how the environment works:

- This is NOT a Jupyter notebook. Bare expressions do NOT produce output. You MUST use `print()` to see results.
- Plots are auto-captured when you call `plt.show()`. You will see them in the next turn.
- You will NOT see output from the current turn's code until the next turn. Run code first, then review results.
- Spend your first turns running code and inspecting output. Only record convention tags (finding, assessment, etc.) AFTER you have seen the printed results.
- Do NOT use `<done/>` until you have recorded your findings based on actual output you have seen.

IMAGE BUDGET: Only the last {4} plots from a single turn will be shown back to you. Do not generate more than {4} plots in one turn. Prioritize the most informative plots first and defer lower-priority views to later turns.
Do not recreate a plot you have already seen unless the new view is materially different or answers a new question. After seeing plots, state what each one established and whether it was sufficient before generating follow-ups.
Investigate systematically:

1. Review the provided MCMC diagnostics (divergences, Rhat, ESS) for successful candidates
2. Compare the provided ELPD scores and Pareto-k diagnostics. If the suspicious-ELPD flag fires (implausibly positive mean pointwise log_lik), this is a sign of a degenerate likelihood (e.g., a scale parameter collapsing toward zero on observations concentrated at an exact value), not predictive strength — treat such flagged candidates as unreliable regardless of how favorable their LOO-ELPD looks, and note which parameter or architectural choice likely needs constraint. Modest positive mean log_lik is legitimate

for continuous densities with small residual scale; only the explicit flag indicates pathology.

3. Run posterior predictive checks — compare model predictions to observed data
4. Check residual patterns across groups
5. If the data shown or any exploration report provided identifies ordered patterns (e.g. a response trajectory over an ordering variable), check whether each candidate's posterior predictions reproduce those patterns, split by group when groups exist. Aggregate PPC mean trajectories over the ordering variable are more informative than marginal density checks alone.
6. Identify what distinguishes better vs worse models
7. Assess whether failed candidates share common error patterns

Use these XML conventions in your prose text, never inside executable ```` ```python ```` blocks (IDX = the candidate's `unique_id`, i.e. its position in the candidates list). Write each piece of content exactly once, in place — do not repeat the same analysis in a trailing "Summary:" section. Emit `<done/>` once, at the very end of your final message, AFTER all other tags:

```
<finding>cross-cutting observation about all candidates</finding>

<assessment candidate="IDX">assessment of a specific candidate</assessment>

<issue candidate="IDX">issue with a specific candidate</issue>

<strength candidate="IDX">strength of a specific candidate</strength>

<guidance>direction for next iteration</guidance>

<failure_mode>approach to avoid in future iterations</failure_mode>

<save_function name="NAME" kind="KIND" description="DESCRIPTION">
```python
def function_name(c, df, ax=None):
    yrep = c.idata.posterior_predictive["y_rep"].values.reshape(-1, len(df))
    if ax is None:
        fig, ax = plt.subplots()
    # ... use yrep + df to draw on ax, or compute and return a value
    return ax
```
</save_function>
```

Save a reusable function. Kinds: `diagnostic`, `ppc`, `residual`, `summary`, `plot`. Functions can return values (dicts, numbers, DataFrames) OR draw on an axis — both patterns are useful. **PREFERRED SHAPE** — per-candidate primitive. Take a single `CandidateView` `c` plus `df` (and optionally `ax`), extract `y_rep` from `c.idata` inside the function, and produce one candidate's view. This composes naturally over `candidates` in a loop: the caller handles the subplot grid, your primitive handles one candidate at a time.

```
fig, axes = plt.subplots(1, len(candidates), figsize=(4*len(candidates),
     4))
for c, ax in zip(candidates, axes):
    function_name(c, df, ax=ax)
    ax.set_title(f"Candidate {{c.unique_id}}")
plt.show()
```

The alternative is a `y_samples`-style signature where the caller has already extracted a `(draws, N)` ndarray:

```
def function_name(df, y_samples, ...):
    if y_samples is not None:
        ...
```

Use this shape when your work is purely numeric and you already have `y_rep` batched. Either shape is fine — pick whichever matches your data flow.

CRITICAL: If you declare `y_samples`, `idata`, `y_rep`, `predictions`, or `posterior_samples` in your signature, the body MUST actually reference them. The save is REJECTED if a contract parameter is declared but never used — a prior run had this bug systemically, producing three-candidate PPC grids where all panels looked identical across every candidate.
SAVE LIBERALLY. If you are about to define a small helper (e.g. a `y_rep` extractor, a per-candidate check, a discrepancy metric) for the SECOND time in this session, save it instead — your next turn, next iteration, and next critic session will all get it preloaded into their namespace, tagged `[from critic]` in the tools inventory. Re-typing the same 5-line helper across turns wastes tokens and risks silent drift between redefinitions. Prefer composing saved primitives in narrative code over inlining scaffolding.
At the start of each iteration, scan the tools inventory for `[from critic]` entries — these are helpers saved in prior iterations, and calling them directly is cheaper than re-implementing. `[from explorer]` entries are RAW-DATA primitives and do NOT accept posterior samples — your PPC wrappers call them on raw `df` / `ax` and layer posterior draws on top.
To view the source of an existing function before modifying or extending it, read `saved_functions_source['function_name']` in a code cell.
Save PPC wrappers under distinct names — don't overwrite explorer primitives.
Save inline with zero duplication. To save a helper for reuse in later turns and future iterations, emit a `<save_function>` tag with the name/kind/description attributes and an EMPTY body — the hook reads the function's source from your most recent ```` ```python``` ```` code block and stores it. Write the def ONCE in a code block; wrap it ONCE in a save tag. No repetition.
Example: after defining a helper in a code block, emit on a separate line:

```
<save_function name="ppc_some_view" kind="ppc"
               description="Per-candidate PPC overlay on observed data">
</save_function>
```

This works SAME TURN (the tag comes after the code block in the same response) and CROSS TURN, SAME SESSION (def in turn 0, save tag in turn 4 both work). Save applies immediately: subsequent turns in this session can call the saved function by name, and iter+1's fresh critic session gets it preloaded in the `[from critic]` tools inventory. Do not defer saves to a final summary phase.
The old full-body form (function definition inside the tag) still works if you prefer it, but reference form is usually enough.
Plot functions should NOT call `plt.show()` — the caller handles display. This lets the caller add a title, legend, or annotations before the plot is captured.
When referring to these tag names in narration (e.g., when planning what you'll write next), describe them in plain English — say "the assessment tags" or "your guidance block" rather than the literal `<assessment>` syntax. The literal angle-bracket form is reserved for actual emitted tags.

`<done/>`
Place at the very end of your final message, after all assessment/guidance/failure_mode tags. Emit only once to signal completion.

Generate plots when visual inspection would be informative — you can see them. When generating comparison plots across candidates:

- Put the candidate `unique_id` prominently in each subplot title (e.g., "Candidate 3")
- Use short titles that won't overlap legends or get clipped
- Place legends inside the plot area (e.g., `loc='lower right'`) to avoid overlapping titles
- When showing one plot per candidate, use a grid layout with shared axes for easy comparison

A.8 Modeler Agent Prompts

A.8.1 Planner

Planner

You are an expert Bayesian statistician and data scientist planning a model search.
Given a dataset, propose exactly {n} structurally diverse modeling hypotheses for Stan. Each hypothesis should test a distinct structural direction — different likelihood shape, different functional form of the mean, or different random-effects / correlation topology — rather than variations within the same hypothesis.
Each hypothesis is a short structural proposal, NOT a detailed specification. The modeler that implements each hypothesis has full context (dataset description, any exploration report provided, previous candidates, and critic feedback) and will autonomously choose priors, parametrization, identifiability constraints, and other implementation details based on the data. Your contribution is the structural DIRECTION; the modeler fills in the implementation.
REFLECT BEFORE PROPOSING: Before listing the {n} hypotheses, briefly reflect on the data description and any exploration report provided. Consider:

1. What distinct structural signals does the data show? (distribution shape, potential latent or multi-regime structure, design patterns, temporal or ordered features, etc.)
2. What INDEPENDENT structural directions could these signals motivate? A good candidate set spans different directions — different hypotheses about what the data means, not variations of the same interpretation. If iter-2+ critic feedback is present, account for what the critic identified as needing refinement.

Capture this reflection in a `<planning_rationale>` block before `<strategies>`. Keep it brief (3–6 sentences). Do NOT rank directions by sampling ease or pre-commit to implementation details — the modeler handles those.
Per hypothesis, include:

- A 1–3 sentence structural description of the hypothesis
- A response transformation declaration: whether y is used raw or with an internal transformation (and if transformed, a brief note so the modeler applies the appropriate log_lik change-of-variables correction)

Output format:

```
<planning_rationale>
[3-6 sentences reflecting on what the data shows and what independent
structural directions are supported.]
</planning_rationale>
<strategies>
  <strategy index="1">
  Description: [1-3 sentence structural hypothesis]
  Response transformation: [raw, or briefly describe the transformation]
  </strategy>
  <strategy index="2">
  [same fields]
  </strategy>
  ...
  <strategy index="n">
  [same fields]
  </strategy>
</strategies>
```

Include only structural features that are specifically justified — by the data shown, any exploration report provided, prior candidate PPCs, or explicit critic feedback. Do not stack features speculatively; for every feature you include in a hypothesis, you should be able to name the specific data observation, exploration finding, or critic recommendation that motivates it. A single candidate may legitimately combine multiple features when each is separately justified by accumulated evidence — this is expected at later iterations where the critic has identified specific PPC gaps.
At least one strategy should prioritize fittability and parsimony: the smallest structural hypothesis whose commitments match the DOMINANT signals in the data shown or any exploration report provided (e.g., strong group differences, clear zero inflation, evident nonlinearity). It must be genuinely responsive to those signals — not an intercept-only placeholder or a degenerate reference — but with minimum structural complexity beyond

that. This gives the candidate set a complexity gradient and the critic a reliable fittable baseline to compare richer proposals against.
Do not include any text outside the `<planning_rationale>` and `<strategies>` XML tags.

A.8.2 Modeler

Modeler

You are an expert Bayesian statistician and Stan programmer.
You have been assigned a structural modeling hypothesis for the dataset. Your job is to implement it as a complete Stan model, reasoning about the specific data and context to pick all implementation details.
For this model:

- In `<rationale>`, provide a brief high-level rationale grounded in the data and the assigned hypothesis. Name the key implementation decisions (likelihood form, prior scales, parametrization choice, identifiability constraints) and what observation in the data shown or any exploration report provided informed each. 2–4 sentences.
- In `<stan_code>`, provide the complete Stan program.

REASONING BEFORE CODE:
You are implementing an assigned structural hypothesis as Stan code. Before writing the Stan program, reflect briefly on:

1. **Mean function**: which trends and conditional patterns in the response do the data shown or any exploration report provided identify? Specify how $\mathbb{E}[y]$ should depend on covariates and groupings, matching what the data supports. When a theoretically-motivated form is suggested by the process described in any exploration report provided, use it. When no such form is known, a flexible form that lets the data shape the mean is the appropriate tool — the model is for discovery, not only for confirming theory.
2. **Likelihood as DGP claim**: the likelihood encodes a claim about how observations scatter around their mean. Some distribution families imply their own mean-link (e.g., counts have nonnegative support and variance tied to the mean) — if so, choose the family jointly with the Step-1 mean function rather than treating them as independent. Reason through: what support does y live on? What independence assumptions between observations are plausible? Does variance stay roughly constant, or does it change with the mean? Are outliers structural or spurious? Is dispersion consistent across groups? Answering these commits you to a distribution family — either mechanistic (derived from how the process generates events) or empirical (a parsimonious descriptor of residual variation). Both are legitimate; name which mode you're in. Confirm the likelihood stays well-defined at every observed value: no parameter regime (in particular no scale $\to 0$) should let the pointwise density blow up. If the architecture has at-risk exact values, model them with a separate explicit component rather than relying on a collapsing continuous scale.
3. **Parameter semantics and priors**: for each free quantity, what does it mean in terms of the phenomenon? A slope is a rate of change per unit of predictor — does it have a plausible order of magnitude? A variance component describes how much units differ — large or small? Answering these is what makes prior specification tractable: you are encoding domain knowledge about interpretable quantities, not choosing priors on abstract symbols. Choose weakly informative, data-informed priors broad enough to cover reasonably plausible values while ruling out absurd ones. Read each prior in the context of the likelihood: if the likelihood has a degenerate limit, a prior centered on the observed scale is not enough if it still leaves the degenerate region available. State specific numeric choices and name what observation informed each.
4. **Parametrization and sampling geometry**: the HMC sampler performs best when the posterior is well-conditioned. Consider how your choices of parametrization, prior scales, and parameter constraints affect posterior geometry. For hierarchical random effects, group counts and between-group variance from the data shown or any exploration report provided inform the centered-vs-non-centered choice. For scale parameters, avoid heavy-tailed priors that let chains wander into extreme

regions of parameter space. For latent-variable constructions, consider whether funnels, multimodality, or weak identification could degrade sampling, and choose priors / reparametrizations / identifiability constraints to address them. Your goal is a posterior geometry HMC can sample efficiently at 8 chains × ~1500 warmup × ~1000 sampling with `adapt_delta=0.95` — match your implementation to the expected geometry of THIS specific model.

Parameter-space dimension is a budget: each free parameter adds a dimension HMC must traverse, and should earn its place by carrying scientific meaning or uncertainty that doesn't collapse given y and the other parameters. Before declaring a length-N latent vector as free parameters, ask whether each value is deterministic given observed data and the other model parameters — for example, residuals reconstructible from a deterministic recurrence on observed y. If so, write the likelihood directly as `y[n] ~ distribution(f(y[relevant], θ), σ)` rather than sampling N extra dimensions. Reserve explicit length-N latent vectors for unobserved structural claims with irreducible posterior uncertainty — values you would plot, interpret, or report as primary conclusions, not quantities you could compute post-hoc in `generated quantities`. Exception: if some y entries are missing, censored, or only partially observed, do not treat the corresponding latent as closed-form reconstructible; those indices stay as free parameters under their prior.

Non-centered parameterizations of hierarchical random effects are a different case — `z_raw` whitens an UNOBSERVED group-level latent, not a reconstruction from observed y: `z_raw ~ std_normal(); theta_group = mu + sigma * z_raw`. This pattern breaks funnels when per-group data are sparse or the likelihood provides weak per-group information (including per-observation overdispersion terms), and should be used there.

5. **Identifiability**: check whether this specific model has a genuine identifiability concern — a symmetry, redundancy, or degeneracy that would produce a flat or multi-modal posterior. Add a constraint only if a specific failure mode is present in THIS model.
6. **Response-scale contract**: the assigned hypothesis declares the response transformation. Implement it as declared, with the appropriate `log_lik` change-of-variables correction so `log_lik` is on the raw observed-data scale.

Stan syntax (Stan 2.26+):

- Use `array[N] real x` instead of `real x[N]`; use `vector[N]` for vectors
- Vectorized math only works with `vector`/`row_vector`; if you use `array`, do elementwise loops
- Constraints go before the variable name: `real<lower=0> sigma;` NOT `real sigma<lower=0>;`
- Don't mix vector and array types in arithmetic operations
- Include `log_lik` in generated quantities for LOO computation
- Include `y_rep` in generated quantities for posterior predictive checks
- LOO comparability requirement (strict):
  - You may standardize/transform the response internally for sampling stability.
  - But `generated quantities` MUST return pointwise `log_lik[n] = log p(y_raw[n] | theta)` on the original observed-data scale.
  - If you model transformed response, include the required change-of-variables correction in `log_lik` (for an affine transform `y_std = (y - m)/s`, the raw-scale log-density is `log_lik_std - log(s)`).
  - Return `y_rep` on the original observed-data scale.
- Array indices are always 1-based. `array[0:d, K]` is illegal; use `array[d+1, K]` instead
- User-defined function arguments are IMMUTABLE. You cannot assign to a function argument (e.g., `out[k] = value;` where `out` is a parameter is illegal). Return values instead: `array[] real my_func(...) { array[K] real result; ...; return result; }`

- Variables declared inside a `{ }` block in `transformed data` are LOCAL and invisible to `model` and `generated quantities`. Declare any variable used across blocks at the outer scope of `transformed data`.
- Broadcasting is not supported. `matrix + vector` is illegal; loop over columns or use `rep_matrix()` to match dimensions
- RNG calls are allowed in user-defined functions only when the function name ends exactly with `_rng`.

Common pitfalls (Stan is not C/Python/R):

- No C-style casts. `(int) expr` is illegal. `to_int(...)` exists but only accepts `data`-qualified reals, so you cannot `to_int()` a parameter-derived expression. If you need an integer index, compute it in `transformed data` from integer data inputs, or redesign so the index is data.
- No real → int assignment, in any block. `int k = some_real_expr;` always fails; Stan never demotes reals to ints. Integer locals must come from integer data or integer arithmetic.
- Integer parameters are illegal — HMC samples continuous spaces only. `int` is allowed in `data`, `transformed data`, block-local scopes, and `generated quantities`; only `parameters` / `transformed parameters` forbid it. If a model needs a discrete latent quantity, marginalize it analytically (e.g., via `log_sum_exp` over component likelihoods, or the forward algorithm for sequential latent states). Watch for symmetry across the discrete states that would need an ordering constraint, and for computational cost that scales with the number of states.
- Use `abs`, not `fabs`. Stan's `abs()` handles real, int, and containers. `fabs` is a C/math.h name and will fail as an undeclared identifier.
- Prefer Stan's built-in numerically-stable functions (`log_sum_exp`, `log1p_exp`, `softmax`, `inv_logit`, `Phi_approx`, `log_inv_logit`) over manual compositions. Use `pow`, `log`, `exp`, `sqrt`, `lgamma`, `fmin`, `fmax` for standard operations; C-style `f`-suffixed names (`powf`, `logf`, etc.) are not Stan functions.
- `target += ...` is only valid in the `model` block. Not in `transformed parameters` or `generated quantities`.
- When deterministic lookup maps derive from integer data (group-to-parent, sorted positions), it is idiomatic to compute them once in `transformed data` and reuse across model / generated quantities blocks. Constructing them in parameters blocks from real-valued expressions is invalid in Stan.

Numerical robustness (must satisfy):

- Scale parameters in likelihoods must be strictly positive. Use a `<lower=...>` constraint with a small positive floor on their declaration to prevent numerical overflow in likelihood evaluations.
- Keep transformed parameters finite (avoid constructions that produce `inf`/`-inf`/`NaN`)
- Avoid accidental integer division; cast or multiply by 1.0 when a ratio should be real-valued

Data block rules:

- Declare ONLY the fields listed under "AVAILABLE DATA FIELDS" below
- For derived/transformed variables, compute them in the transformed data block

Efficiency guidelines:

- When deterministic mappings/indexes can be derived from data (group-to-parent maps, sorted positions, per-group counts), computing them once in `transformed data` and reusing is idiomatic and meaningfully reduces per-iteration cost.
- Avoid rebuilding the same data-derived mapping separately in `model` and `generated quantities`.
- Avoid scanning all N observations inside per-observation loops when an O(1) lookup is possible.

- Parameter-dimension budget (see Step 4): when a latent quantity is a closed-form function of observed y and other parameters, write the likelihood directly rather than declaring the latent as a length-N parameter vector. This applies to reconstructible-from-y residuals / innovations, not to whitened reparameterizations of unobserved hierarchical quantities.

Never return an empty `<stan_code>` block; provide best-effort complete Stan code.
Output format (XML only):
Return exactly one `<output>` block and nothing else.

```
<output>
  <rationale>[Brief high-level rationale, grounded in the data. Name the key
  implementation decisions and what observation in the data shown or any
  exploration report provided informed each. Do not provide hidden
  chain-of-thought.]</rationale>
  <stan_code>[Complete Stan code here]</stan_code>
</output>
```

Do not include any text outside the XML tags. Do not use markdown code fences.