Contributing to Stan

A Developer's Guide to the Core, Interfaces, and First Contributions

Florence Bockting

2026-08-17

Table of contents

Goals

This material is for anyone interested in contributing to Stan but unsure where to start. By the end of the tutorial, you will be able to:

- **Place yourself in the ecosystem**: Know which projects make up Stan, from the compiler and math library through `CmdStan` to the language interfaces and workflow packages, and work out which repository a given contribution belongs in.
- **Pick a contribution that fits you**: Recognise the full range of ways to help, from answering questions on Stan Discourse, to filing bug reports and feature requests, improving documentation, writing case studies, and contributing code.
- **Reach out effectively**: Open a GitHub issue or pull request that maintainers can act on, and know when Discourse is the better place to start the conversation instead.
- **Find the right information**: Match a question to the right resource among user guides, API documentation, case studies, and articles.
- **Follow a concrete first contribution**: The checklist walks step by step through the practicalities, with a separate checklist for the R packages, the ArviZ Python packages, and the Stan Math C++ library. These are three genuinely different processes — different base branch, test runner, and changelog handling — so the chapter ends with a table showing where they diverge.
- **Use the contributor skills**: This repository ships a set of Agent Skills for the Stan contribution workflows. They work with an LLM agent, and they work on their own as human-readable recipes, issue templates, and a pre-pull-request check script.
- **See it done end to end**: Two walkthrough chapters follow a real contribution from finding an issue to writing the pull request, including the steps that commonly go wrong — one to an R package, `posterior`, and one to a Python package, `arviz-stats`. Both also show the Agent Skills in this repository being used along the way.

You do not need to be a professional software developer or statistician, and you do not need prior open-source experience.

1 What is Stan?

By the end of this section, you'll understand that Stan is a probabilistic programming language for specifying statistical models, an open-source community, as well as a broader software ecosystem.

1.1 Language, Community, Ecosystem

Stan is ...

- an open-source community consisting of people from a variety of different backgrounds (Cognitive Science, Computer Science, Statistics, etc.).
- an open-source ecosystem consisting of a core library, interfaces for multiple programming languages, and a growing collection of contributed packages.
- a probabilistic programming language for specifying statistical models.

1.1.1 An open-source community

We communicate via:

- Slack
  - different channels (e.g., #general, #help)
  - mainly developer discussions
- Discourse
  - questions, discussion, and announcements related to Stan
  - for both users and developers
- GitHub
  - code repositories
  - issue tracker for reporting bugs and requesting features
  - pull requests for contributing code, documentation, examples, etc.

- Stan Website
  - documentation, tutorials, case studies, etc.

1.1.2 An open-source ecosystem

Multiple packages/libraries that work together to provide a complete probabilistic programming environment

More on this later!

1.1.3 A probabilistic programming language

A probabilistic programming language for specifying statistical models.

```
data {
  int N;   // number of observations
  int J;   // number of students
  int K;   // number of items on exam
  array[N] int<lower=0, upper=J> student;
  array[N] int<lower=0, upper=K> item;
  array[N] int<lower=0, upper=1> y;
  vector[J] x;
  real mu_mu_a, mu_mu_b;
  real<lower=0> sigma_mu_a, sigma_mu_b, mu_sigma_a, mu_sigma_b;
}
transformed data {
  vector[J] x_adj = (x - mean(x))/sd(x);
}
parameters {
  real mu_a, mu_b;
```

```
  real<lower=0> sigma_a, sigma_b;
  vector<offset=mu_a, multiplier=sigma_a>[K] a;
  vector<offset=mu_b, multiplier=sigma_b>[K] b;
}
model {
  a ~ normal(mu_a, sigma_a);
  b ~ normal(mu_b, sigma_b);
  mu_a ~ normal(mu_mu_a, sigma_mu_a);
  mu_b ~ normal(mu_mu_b, sigma_mu_b);
  sigma_a ~ exponential(1/mu_sigma_a);
  sigma_b ~ exponential(1/mu_sigma_b);
  y ~ bernoulli(0.25 + 0.75*inv_logit(a[item] + b[item] .* x_adj[student]));
}
```

2 The Stan Ecosystem

By the end of this section, you'll understand that the Stan ecosystem is a stack of related projects mainly developed under stan-dev, ranging from the compiler and math library to `CmdStan`, language interfaces, and workflow packages. You'll be able to place a typical modeling task within that stack and identify which layer each piece belongs to.

2.1 A minimal Stan workflow

We start our workflow by writing a minimal Stan model in a `.stan` file.

Listing 2.1 `R · Stan model`

```
stan_code <- "
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  mu ~ normal(0, 10);      // prior
  sigma ~ exponential(1);  // prior
  target += normal_lpdf(y | mu, sigma);  // likelihood
}
generated quantities {
  vector[N] log_lik;
  for (n in 1:N) {
    log_lik[n] = normal_lpdf(y[n] | mu, sigma);
  }
}
"
writeLines(stan_code, "model.stan")
```

Additionally, we provide some data for the input and save it as a JSON file.

Listing 2.2 `R · data`

```
library(cmdstanr)

stan_data <- list(N = 20, y = rnorm(20, mean = 5, sd = 2))
cmdstanr::write_stan_json(stan_data, "data.json")

> stan_data
$N
[1] 20

$y
 [1] 4.543589 4.899750 4.081620 8.698614 4.827003 2.903813 8.659615 5.165155
 [9] 5.148127 1.206675 2.368544 7.873981 5.071258 5.322390 6.913431 7.336083
[17] 3.462279 3.717310 1.620141 3.577615
```

2.1.1 Language interfaces

Next we can compile and sample from the Stan model. To do this, we use an interface, such as `cmdstanr` for R or `cmdstanpy` for Python, that wraps `CmdStan`.

`CmdStan` invokes

- `stanc3` to transpile the Stan model into a C++ file, which is then compiled and linked against
- the `math` library (implementing built-in functions for distributions, linear algebra, ODE solvers, etc., along with automatic differentiation) and
- the `stan` library (implementing the inference algorithms, such as HMC/NUTS).

The result is a standalone executable that the interface then runs to produce posterior samples.

Listing 2.3 `R · cmdstanr`

```r
library(cmdstanr)
library(bayesplot)
library(loo)

# compile Stan model
mod <- cmdstanr::cmdstan_model("model.stan")

# sample from the model
fit <- mod$sample(data = "data.json", chains = 4, iter_sampling = 1000)

# get draws
draws <- fit$draws(variables = c("mu", "sigma"))

# plotting and cross-validation
bayesplot::mcmc_trace(draws)
loo::loo(fit$draws("log_lik"))
```

2.1.1.1 The R interface `cmdstanr`

Compile and sample from the Stan model in R with `cmdstanr`. Use the fitted model to extract posterior draws, plot MCMC trace plots with `bayesplot`, and run approximate leave-one-out cross-validation with `loo`.

2.1.1.2 The Python interface `cmdstanpy`

Compile and sample from the Stan model in Python with `cmdstanpy`. Use the fitted model to extract posterior draws, plot MCMC trace plots, and run approximate leave-one-out cross-validation using `arviz`.

2.1.1.3 The Command-line interface `CmdStan`

`cmdstanr` and `cmdstanpy` are thin language wrappers around `CmdStan`, the command-line interface to Stan. `CmdStan` itself invokes `stanc3` to compile a `.stan` file into a standalone executable, which can be run directly from the command line to sample from (or optimize, etc.) the model.

Listing 2.4 Python · cmdstanpy

```python
from cmdstanpy import CmdStanModel
import arviz as az

# compile Stan model
mod = CmdStanModel(stan_file="model.stan")

# sample from the model
fit = mod.sample(data="data.json", chains=4, iter_sampling=1000)

# get draws
idata = az.from_cmdstanpy(fit, log_likelihood="log_lik")

# plotting and cross-validation
az.plot_trace(idata, var_names=["mu", "sigma"])
az.loo(idata)
```

Listing 2.5 Bash

```bash
# run from the CmdStan directory
make model

./model sample num_chains=4 num_samples=1000 \
  data file=data.json \
  output file=output.csv

$CMDSTAN/bin/stansummary output_*.csv
$CMDSTAN/bin/diagnose output_*.csv
```

2.1.2 Higher-level modelling frameworks

Instead of writing a Stan model from scratch, you can use higher-level modelling frameworks that let you specify your model using formula syntax. Two example frameworks are

- brms, which dynamically generates and compiles custom Stan code for your specified model, and
- rstanarm, which fits your model using a set of pre-compiled Stan programs for common model families (avoiding compilation time, at the cost of some flexibility).

Listing 2.6 `R · brms`

```r
library(brms)

dat <- data.frame(y = jsonlite::fromJSON("data.json")$y)

fit <- brms::brm(
  formula = y ~ 1,
  data = dat,
  family = gaussian(),
  prior = c(
    prior(normal(0, 10), class = Intercept),
    prior(exponential(1), class = sigma)
  ),
  chains = 4,
  iter = 2000
)
```

2.2 The Stan ecosystem and its libraries

To summarize what we have seen so far, we can outline the different layers in the Stan ecosystem and how they relate to one another.

Different layers in the Stan ecosystem

- **Layer 1: Compiler & math**: `stanc3` · `stan` · `math`
  - Contribute to the fundamental building blocks of the Stan language: compiler transpilation, autodiff, or core inference algorithms.
- **Layer 2: Command-line engine**: `cmdstan`
  - Contribute to the command-line interface that drives compilation and execution.
- **Layer 3: Language interfaces**: `cmdstanr` · `cmdstanpy` · `rstan` · …
  - Contribute to environment-specific host language wrappers (R, Python, Julia, etc.) that interface with Stan.
- **Layer 4: Workflow & analysis**: `posterior` · `bayesplot` · `loo` · `priorsense` · …
  - Contribute to downstream packages used among others for visualization, posterior diagnostics, cross-validation, and sensitivity analysis.

Listing 2.7 `R · rstanarm`

```r
library(rstanarm)

dat <- data.frame(y = jsonlite::fromJSON("data.json")$y)

fit <- rstanarm::stan_glm(
  formula = y ~ 1,
  data = dat,
  family = gaussian(),
  prior_intercept = normal(0, 10, autoscale = FALSE),
  prior_aux = exponential(1, autoscale = FALSE),
  chains = 4,
  iter = 2000
)
```

2.2.1 Compiler, math, and CmdStan

Package	Description	Relations	Language
math	Autodiff library for distributions, linear algebra, and gradients	Used by generated model code	C++
stanc3	Compiler that translates `.stan` to C++	Feeds `cmdstan` and `rstan`	OCaml
stan	Inference algorithms and services (HMC/NUTS, etc.)	Built on `math`; used via `cmdstan` / `rstan`	C++
cmdstan	Command-line engine to compile and run Stan models	Wraps `stanc3` + `stan` + `math`; backend for `cmdstanr` / `cmdstanpy`	C++ / CLI

2.2.2 Language interfaces

Package	Description	Relations	Language
cmdstanr	R wrapper around CmdStan; returns `CmdStanMCMC`	Calls `cmdstan`; used with `posterior`, `bayesplot`, `loo`	R
cmdstanpy	Python wrapper around CmdStan; returns `CmdStanMCMC`	Calls `cmdstan`; used with `arviz`	Python
rstan	R interface that embeds Stan in-process; returns `stanfit`	Alternative to CmdStan; used by `rstanarm` and `rstantools` packages	R

2.2.3 Workflow, modeling, and packaging

Package	Description	Relations	Language
brms	Flexible multilevel / distributional regression via formulas; returns `brmsfit`	Generates Stan; backends `cmdstanr` / `rstan`; used with `loo`, `bayesplot`	R
rstanarm	Precompiled applied regression models; returns `stanreg`	Built on `rstan`; used with `loo`, `bayesplot`, `projpred`	R
posterior	Tools to manipulate and summarize draws (R-hat, ESS, etc.)	Shared draws API for `cmdstanr` and analysis packages	R
bayesplot	ggplot2 plots for MCMC diagnostics, PPC, and posteriors	Uses draws from interfaces / `posterior`	R

2.2.4 Workflow, modeling, and packaging (continued I)

Package	Description	Relations	Language
loo	Approximate LOO-CV (PSIS-LOO), WAIC, and model weights	Needs pointwise `log_lik` from fitted models	R

Package	Description	Relations	Language
arviz	Diagnostics, plots, and model comparison in Python; returns `xr.DataTree`	Python counterpart to `posterior` / `bayesplot` / parts of `loo`	Python
priorsense	Prior and likelihood sensitivity analysis (power-scaling)	Uses `posterior`; related to PSIS methods in `loo`	R
projpred	Projection predictive variable selection	Uses `stanreg` (`rstanarm`) or `brmsfit` (`brms`) as reference	R

2.2.5 Workflow, modeling, and packaging (continued II)

Package	Description	Relations	Language
shinystan	Interactive Shiny GUI for MCMC diagnostics	Uses `stanfit` / `stanreg`; overlaps with `bayesplot`	R
rstantools	Tools to ship Stan models inside R packages; models typically return `stanfit`	Scaffolding used by packages such as `rstanarm`	R
posteriordb	Database of models, data, and reference posteriors	Shared resource for testing, teaching, and CI	R / Python / data

3 How to contribute to Stan?

By the end of this section, you will know the different ways of contributing to Stan, from engaging on Stan Discourse, over filing bug reports and feature requests, writing documentation and case studies, to contributing code. You will know how to pick a contribution path that fits you, write a clear GitHub issue or pull request, and follow project norms.

> **!** Important
>
> You don't need to be a professional software developer or statistician in order to contribute to Stan!

3.1 Types of contributions

There are many ways to contribute to Stan, including:

- asking and answering questions on Stan Discourse
- reporting problems when interacting with the software or documentation
- requesting new features or improvements
- providing feedback on documentation
- contributing with a case study of your specific research field
- contributing code

3.2 Typical workflows for contributing

3.2.1 Participate in the Stan Discourse

- Stan Discourse is the main forum for users and developers to ask questions, discuss issues, and share ideas.
- We aim to keep this a friendly and welcoming space, guided by our:
  - Code of Conduct
  - Discourse guidelines

- We also discuss topics on Slack, but we encourage users to ask questions on Stan Discourse instead as it's open to everyone, so answers can help other community members too.

3.2.2 Feature requests and bug reports

- Go to the GitHub repository you want to contribute to.
- Get familiar with its Issues and PRs (e.g., loo).
- If you're a new contributor:
  - Look for labels like "help wanted" or "good first issue."
  - Avoid issues that involve refactoring or larger changes.
  - When in doubt, ask in the issue whether it's a good one to work on.
- To open a new Issue:
  - First check whether it already exists, to avoid duplicates.
  - Some tips for writing good Issues...

Writing good Issues: Bug Report

- Title: One sentence, specific, searchable.
- Body
  - What happened vs what you expected
  - Minimal reproducible example (i.e., small model/script/data that triggers the problem)
  - (Optional) Environment: OS, hardware if relevant
  - Exact steps to reproduce the problem.
  - Logs / error output (if possible as text in a code block, not a screenshot, so it's searchable)
- Examples:
  - bayesplot
  - brms

Writing good Issues: Feature Request

- Title: One sentence, specific, searchable.
- Body
  - describe the problem or limitation
  - (optional) propose solution
  - (optional) note scope of Issue: small, self-contained or does it open a design discussion?
- Examples:
  - bayesplot

Additional references:

- Mozilla's guide to writing good bug reports
- "How to Report Bugs Effectively" by Simon Tatham
- R package "reprex" (tidyverse): runs your code, captures output, and formats it (code + results) as ready-to-paste Markdown for GitHub issues

---

3.2.3 Feedback on documentation

- Clarifying confusing sections
- Adding examples or code snippets
- Fixing inaccuracies (e.g., when current behavior deviates from documented behavior)
- Flagging content that is missing or unclear
- Fixing typos, grammar, and broken links

3.2.4 Contributing a case study or tutorial

- list of case studies in Stan
- blog post by Bob Carpenter on expanding the Stan Users Guide

Checklist for contributing a case study

1. **Initial contact**
   - post case study or tutorial idea in the Stan Discourse first
2. **Content requirements**
   - write it as a narrative document, not just code

- use a reproducible-notebook format
- note exact software and compiler versions used
- fix random seeds so results are exactly reproducible

Checklist for contributing a case study (continued)

3. Licensing

- license code under an open source license (e.g., BSD or GPL)
- license text as Creative Commons (CC-BY is most common)
- author retains copyright (but Stan needs to be permitted to host and redistribute)

4. Metadata for published case study

- Title and Author(s) (optional affiliation)
- Short abstract/description (in 2-4 sentences, what problem is solved and why it matters)
- Keywords (3-6 terms for discoverability)
- Source repository link (e.g., a public GitHub repo containing the actual notebook/knitr source, separate from the rendered HTML)
- Dependencies list (package/library versions used)
- Rendered HTML version (the actual output others will read)

3.2.5 Contributing Code

Some packages have templates others don't, but overall a good PR description should incorporate the following information:

Writing a good PR description

Before you start:

- Have you read the contributing guidelines and AI Contribution Policy?
- Contributing to Stan core libraries (e.g., `stan`, `math`, `stanc3`)? Checkout Developer Wiki?
- Is your PR small and focused on one logical change?
- Not ready for review yet? Open it as a **Draft PR** so reviewers know to hold off.

Title: One sentence, specific, searchable.

- Good: `Fix incorrect ESS calculation for thinned chains`

- Avoid: `Bug fix`

Writing a good PR description (continued I)

Body:

- Link PR to Issue via GitHub keywords like `Fixes #123` or `Closes #123`

Description

- Describe what has been changed and why.
- Be concise and use your own words (try to avoid using AI-generated text).
- Add any uncertainties or open questions you have here as well.

Breaking changes: Does this PR change existing behavior, APIs, or output? If yes, describe the impact.
Tests: (if applicable) Which tests have you added?
Documentation / Vignettes: (if applicable) Have you added documentation?

Writing a good PR description (continued II)

Checklist

- Style is consistent with existing code and docs
- Documentation or vignette renders locally
- Tests and `devtools::check()` pass
- `NEWS.md` entry added
- If you used an AI assistant, please disclose and review changes critically
- For the full checklist, see checklist

TODOs: List any remaining tasks you need to complete before the PR is ready to be merged.
Examples:

- bayesplot
- brms

3.2.6 Some reflections on the use of AI-assisted development

- Stan AI Contribution Guideline
- Use your own words when writing the PR description and comments in a PR review process
- Review all changes made by an AI assistant critically and ask yourself:
  - Is this change correct?
  - Do I need this change?
  - Is this change in the scope of the PR?
- When in doubt about something, ask rather the Stan developers than the AI assistant

4 Where to find information?

By the end of this section, you'll have a good sense of the main Stan resources including user guides, tutorials, case studies, and API documentation. We'll walk through how to match your question to the right resource.

4.1 Different layers of documentation

The Diátaxis framework provides a useful way to think about the different types of content that make up documentation. It divides documentation into four categories:

Type	Focus	Stan examples
Reference	What is this?	Function/package index, Stan math docs, Stan reference manual
How-to guides	How do I solve this?	Vignettes, case studies
Tutorials	How do I get started?	User's Guide walkthroughs, example-models
Explanation	Why is it like this?	Wiki, discussions on Discourse

The four are not a ranking, and no single document has to serve all of them. Most frustration with documentation comes from looking for one kind and finding another.

4.1.1 Reference ("What is this?")

Dry, complete, factual descriptions of the machinery. Structured for lookup rather than reading front to back.

- *"What is a valid Stan program? What are blocks, types, and constraints, and how do they behave?"*
  - see Stan reference manual
  - grammar/rules of the Stan language
  - analogous to a grammar book

- *"What does `normal_lpdf` in `stan_code` do, and what arguments/types does it expect?"*
  - see Functions reference
  - built-in vocabulary of the Stan language
  - analogous to a dictionary
- *"What does `loo()` in the loo R package do, and what arguments does it take?"*
  - see Package index (API) of corresponding downstream package
  - Examples: loo, ShinyStan
  - Analogous to a manual for a specific translation tool

4.1.2 How-to guides ("How do I solve this?")

Recipes for a specific task, aimed at someone who already knows the basics and has a goal in mind. In the Stan ecosystem these are mostly vignettes, published on each package's website.

- *"How do I fit a Stan model?"*
  - see Stan user guide
- *"How do I plot MCMC draws with bayesplot?"*
  - see Vignette in bayesplot
- *"How do I perform leave-one-out cross validation with loo?"*
  - see Vignette in loo
- *"How do I estimate multivariate models in brms?"*
  - see Vignette in brms

4.1.3 Tutorials / Case Studies ("How do I get started?")

Guided lessons that take a newcomer through a complete, working example. The goal is not to solve your problem but to build familiarity. The path is chosen for you and is meant to succeed.

- *"I have never used Stan. Where do I start?"*
  - see Tutorials
- *"How do these pieces fit into a full Bayesian workflow?"*
  - see case studies on the Stan website, in the Bayesian Workflow book, and on Aki Vehtari's website

4.1.4 Explanation ("Why is it like this?")

Context, design decisions, and alternatives considered. This is the material you read away from the keyboard, and the only category where opinion and history belong.

- *"What is the reasoning behind a method, and how was it validated?"*
  - see the paper accompanying the package, e.g. posterior, brms, bayesplot, ArviZ
- *"Why is Stan designed this way?"*
  - see Discourse discussions
  - see Wiki

---

5 Checklist

The Stan ecosystem does not have one contribution process. The R packages, the ArviZ Python packages, and the C++ core each have their own, and they differ in ways that will trip you up if you carry habits between them: which branch you start from, whether an issue is required, what runs the tests, and who writes the changelog. Pick the checklist that matches your target repository, and see the three processes at a glance for where they diverge.

Contributing to R packages (click to expand)

Getting started

- Check open issues or open a new one to discuss your proposed change before starting significant work (for bigger changes; typo/doc fixes can skip this)
- Check the repository for a `CONTRIBUTING.md` with package-specific guidance

AI contribution policy

- if you used AI tools (e.g. Claude, ChatGPT, GitHub Copilot) to help write your contribution, review the Stan project's AI Contribution Policy before opening a PR
- posterior, bayesplot, loo, and cmdstanr state in their `CONTRIBUTING.md` that all contributions must follow it, so disclose AI assistance in your PR description
- other packages (e.g. rstanarm, brms) do not reference it; disclosing there is good practice rather than a requirement
- you remain responsible for understanding, testing, and being able to explain any AI-generated code you submit

Fork and clone

- raw git: fork on GitHub, then `git clone https://github.com/your-username/repository-name.git`
- usethis: `usethis::create_from_github("stan-dev/repository-name", fork = TRUE)`

Create a branch for your changes

- raw git: start from the main/master branch: `git checkout -b fix/123-short-description`
- usethis: `usethis::pr_init("brief-description-of-change")`

Make your changes on this branch

- keep your branch up to date with upstream
- install development dependencies `devtools::install_dev_deps()`
- `devtools::check()` at the end compiles the package, so you need a compiler: RTools on Windows, the Xcode command line tools (`xcode-select --install`) on macOS. Check this now, not after you have written the change. Windows users: the R installer also leaves `Rscript` off your `PATH`
- debug with `devtools::load_all()` and `debug(new_function)`
- if you add a new function:
  - add tests and run them with `testthat::test_file()` or `testthat::test_dir()`
  - add documentation to new functions and examples
  - render the documentation with `devtools::document()`
- if you add a new vignette:
  - ensure vignette is rendered locally with `devtools::build_vignettes()`
- ensure you follow the style of the existing code/docs

Contributing or changing plots (bayesplot in particular)

- bayesplot uses `vdiffr` for visual regression testing. In visual regression testing plots are compared against reproducible SVG snapshots rather than just checked for errors
- for any new plotting function or a change to an existing plot's appearance, add a graphical test using `vdiffr::expect_doppelganger("descriptive-title", plot_object)` in the relevant test file
- run `devtools::test()`, new or changed snapshots will be flagged
- review flagged snapshots with `testthat::snapshot_review()`, which opens an interactive Shiny app showing old vs. new SVG side by side
- only accept a snapshot change if the visual difference is the one you intended
- SVG snapshots record text as glyph positions, so they depend on the fonts installed on the machine that rendered them. Commit the ones you write, but if snapshots you never touched start failing, suspect fonts before you suspect your plot

Update NEWS.md

- add a bullet for your change, placed just below the first header in NEWS.md
- follow the existing formatting and tone used in that file (check recent entries for the convention)

Finalize your changes

- Add and commit your changes in reasonable chunks `git add <files>` and `git commit -m "short description of your changes"`
- cheatsheet for conventional git commit types
- run the full test suite with `devtools::test()`
- run the full check with `devtools::check()`
- push your branch
  - raw git: `git push origin fix/123-short-description`
  - usethis: `usethis::pr_push()` (opens the PR flow in your browser for you)

Open a PR

- add a detailed description of your changes (see How to write a good PR description)
- expect CI to run additional checks automatically

i Contributing to ArviZ (Python) (click to expand)

ArviZ is a metapackage: most contributions go to one of the sub-packages, not to `arviz-devs/arviz`. The ArviZ contributing guide is the authority; below is the short version.

Getting started

- pick the right repository:
  - `arviz-base` (how data is *represented*),
  - `arviz-stats` (computing a *number*),
  - `arviz-plots` (*drawing* something)
- comment on the issue before starting work
- check the feature does not already exist under another name
- read the pull request tutorial and the sub-package's own contributing pages under `python.arviz.org/projects/`

AI assistance (recommended practice)

- ArviZ has **no** AI contribution policy
- disclosing AI assistance in the PR description is still good practice, and reviewers appreciate knowing; the AI Contribution Policy is a reasonable model to follow voluntarily
- either way, you remain responsible for understanding, testing, and being able to explain every line you submit

Fork, clone, and add the upstream remote

- `gh repo fork arviz-devs/arviz-stats --clone`

- `git remote add upstream git@github.com:arviz-devs/arviz-stats.git`

Set up the environment

- `python3 -m venv .venv && source .venv/bin/activate` — on Windows, `python -m venv .venv` then `.venv\Scripts\Activate.ps1`
- `pip install tox` and `pip install -e .`
- `tox list -m dev` shows the available development tasks
- `pip install -e ".[test]"` install test dependencies if you want to run `pytest` directly
- if your change spans repositories, install both editable in dependency order — `arviz-plots` declares the other two as *git* dependencies, so a plain `pip install -e ./arviz-plots` silently downloads a fresh `arviz-stats` from GitHub and ignores your local changes

Install the pre-commit hooks

- `pip install pre-commit pylint && pre-commit install`
- `pre-commit run --all-files`
- the hook config includes `no-commit-to-branch --branch main`, so the repository will physically refuse a commit on `main`

Create a branch and make your changes

- `git checkout -b <new-branch-name>` (create a new branch and switch to it)
- Code contribution: place code at the right architectural layer
- write numpydoc docstrings
- for user-facing changes, add inline examples, `See Also` links, and `References` if you are implementing a published method

Add tests

- tests are parametrised with `@pytest.mark.parametrize` and use shared fixtures

Run the tests and checks

- `tox -e full` runs the tests
- `tox -e check` runs style and lint
- in `arviz-stats` the test environments are split by dependency set:
  - `tox -e full`,
  - `tox -e minimal`,
  - `tox -e xarray`,
  - `tox -e nightlies` etc.
  - `tox list` shows them all
- **a missing dependency skips a test, it does not fail it**

  - Run `pytest tests/test_metrics.py -q -rs` to see the module that failed to import

- trust `tox -e full`: it sets `ARVIZ_REQUIRE_ALL_DEPS=TRUE`, which turns skips into errors, so a failure there is a real failure.
- to debug, put a `breakpoint()` in the function and run the test with `pytest tests/test_metrics.py --pdb` (the R equivalent is `debug()`)

Rebase, push, and open a PR

- `git fetch upstream && git rebase upstream/main`
- `git push -u origin add-brier-score`
- mark the PR `[WIP]` if it is not finished
- **do not edit `CHANGELOG.md`** — it is generated at release time from merged PR titles

ℹ Contributing to the Stan Math library (C++) (click to expand)

The developer process overview and the Math contributor help pages are the authorities for this contribution workflow.
Branches come off `develop`, **not** `main` or `master`; every pull request must correspond to an issue; and branch names are prescribed.

> **Important**
>
> This checklist is not yest tested and is work in progress.

Getting started

- open an issue first — issues are reserved for bugs and feature requests "defined well enough for a developer to tackle", and vague ones get closed and moved to the forums
- prefer several small issues over one large one
- general questions go to the Developers tag on Discourse, not the issue tracker
- for a new distribution, get it working as a user-defined function in the Stan language and post it on Discourse *before* writing any C++
- read `.github/CONTRIBUTING.md` and the coding style and idioms

AI contribution policy

- all contributions to Math must follow the AI Contribution Policy; this is stated in the repository's own contributing guide

Fork, clone, and configure git

- `gh repo fork stan-dev/math --clone`
- `git remote add upstream https://github.com/stan-dev/math.git`

- `git config push.default simple` and `git config merge.ff false`
- install the pre-push hooks shipped in `hooks/`, which block direct pushes to `master` and `develop`

Create a branch from `develop`

- `git checkout develop && git pull upstream develop`
- features: `git checkout -b feature/issue-1234-short-description`
- bug fixes: `bugfix/issue-1234-short-description`, off the latest hotfix
- `develop` is the integration branch and must always be releasable; `master` holds tagged releases only

Write the function

- a new function starts as one generic implementation in `prim/fun`, written so it accepts `double`, reverse-mode `var`, and forward-mode `fvar` scalars
- add specializations in `rev`, `fwd`, `mix`, or `opencl` only when there is a reason, such as an analytic gradient that beats the autodiff one
- every function outside the `internal` namespace must support higher-order autodiff
- use the library's patterns: `require_*` type traits to constrain templates, `to_ref()` before coefficient access on Eigen expressions, `value_type_t` rather than hard-coded `double`, and `.coeff()`/`.coeffRef()` for unchecked access
- document with doxygen — `make doxygen` must produce **zero** warnings

Add tests

- **bug fix**: at least one test that fails before the patch and passes after — write and commit that test *first*
- **new feature**: at least one test showing expected behaviour *and* one showing behaviour on error; expect the reviewer to ask for more
- new functions are verified with the `expect_ad()` framework, which checks values and derivatives up to third order against finite differences, over every combination of primitive and autodiff argument types
- tests go in `test/unit/math/mix/fun/<name>_test.cpp`
- loosening a tolerance to make a test pass is a red flag — it usually means the derivative is wrong

Run the required checks

All five must pass before you open the pull request:

- `./runTests.py test/unit` — unit tests
- `make test-headers` — every header compiles standalone
- `make test-math-dependencies` — no forbidden cross-folder includes
- `make doxygen` — documentation builds, 0 warnings

- `make cpplint` — style, 0 new errors

While iterating, run only what you touched (`./runTests.py test/unit/math/prim/fun/foo_test.cpp`) and the full suite once before pushing. For a debug build, add `DEBUG = 1` to `make/local`.

> **Warning**
>
> These are the paths in `stan-dev/math`. In `stan-dev/stan` the tests live under `src/`, so it is `./runTests.py src/test/unit`.

> **On Windows**
>
> Two of the five commands are spelled differently. The runner is `python runTests.py test/unit` — the `./` form relies on a shebang, which Windows does not read. `make` is `mingw32-make`, from RTools, which also supplies the compiler and `doxygen`. Check that you have all of this *before* you write the C++, not when you reach this step.

Open a PR

- base repository `stan-dev/math`, **base branch `develop`**
- the template asks for Summary, Tests, Side effects, Release notes, and a checklist
- **name the copyright holder** — yourself, or your university or employer; submitting agrees to license the code under BSD 3-clause and the documentation under CC-BY 4.0
- write the **release note** for a user: it goes into the release notes
- fix review feedback on the same branch; do not open a replacement PR
- anyone who reads C++ fluently can review, so reviewing is itself a good way to contribute

5.1 The three processes at a glance

	R (e.g. `posterior`)	Python (`arviz-stats`)	C++ (`math`)
Base branch	`main` / `master`	`main`	**`develop`**
Issue required	recommended	claim it by comment	**yes, one per PR**
Branch name	free	free	**`feature/issue-<n>-desc`**
Environment	`devtools::install_dev_deps()`	`uv sync`	`make`, no environment manager
Pinned tooling	roxygen2 version in `DESCRIPTION`	ruff/pylint hashes in pre-commit	compiler and `lib/` versions

	R (e.g. `posterior`)	Python (`arviz-stats`)	C++ (`math`)
Documentation	roxygen2 → `man/*.Rd`	numpydoc → `.pyi` stubs	doxygen, 0 warnings
Tests	`testthat` (+ `vdiffr`)	`pytest`, parametrised	`expect_ad()` vs finite differences
Run tests	`devtools::test()`	`tox -e full`	`./runTests.py test/unit`
Full check	`devtools::check()`	`tox -e check`	five separate `make` targets
Style	conventional, reviewed by humans	enforced by pre-commit	`make cpplint`, 0 new errors
Changelog	edit `NEWS.md` yourself	**generated from your PR title**	"Release notes" field in the PR
Licensing step	—	—	**name the copyright holder**

6 The Stan contributor skills

This repository ships a `skills/` folder next to the book. The book explains *why* a contribution looks the way it does; the skills carry the commands, the order, and the traps.

They are written as Agent Skills, an open standard, so an agent such as Claude Code can load them. But every file in them is plain Markdown, so you can also just read them.

> ⚠ Still a prototype
>
> These skills were written for this tutorial and have not been used by many contributors yet. Use them with care, check what they tell you against the repository you are working in, and report problems at stancon-contributing.

6.1 The seven skills

Skill	Use it when you want to
`stan-contributing`	Start somewhere — pick a path and the right repository
`stan-r-package-contribution`	Change an R package: posterior, bayesplot, loo, cmdstanr, brms, …
`stan-python-package-contribution`	Change a Python package: arviz-base, arviz-stats, arviz-plots, cmdstanpy
`stan-math-contribution`	Change C++ in stan-dev/math or stan-dev/stan (not tested, work in progress)
`stan-issue-and-pr`	Write a bug report, feature request, or pull request description
`stan-case-study`	Write and submit a case study or tutorial
`stan-find-docs`	Find where something about Stan is documented

`stan-contributing` is the entry point and routes to the others. The rest also fire on their own when you describe what you are doing.

6.2 What is inside one

A skill is a folder. The only required file is `SKILL.md`: a short YAML header with a `name` and a `description`, then the procedure in Markdown.

```
stan-r-package-contribution/
  SKILL.md          the procedure, step by step
  references/       worked-example.md, plots-and-vdiffr.md, vignettes.md
  scripts/          check_environment_setup.R
```

- **`SKILL.md`** is the recipe. An agent reads the `description` to decide when the skill is relevant, and only then loads the rest.
- **`references/`** hold the longer material a step points to — for example `worked-example.md`, a transcript of a real contribution to posterior #238 with the commands that were actually run and the parts that went wrong.
- **`templates/`** are files to copy out and fill in: `bug-report.md`, `feature-request.md`, `pr-description.md`.
- **`scripts/`** are runnable. `check_environment_setup.R` checks seven things before you open a pull request — roxygen2 pin, dev packages, clean tree, branch, `gh` authentication — and changes nothing.

6.3 Two ways to use them

6.3.1 With an agent

Copy the folders into your agent's skills location and restart the session:

```
git clone https://github.com/florence-bockting/stancon-contributing.git
mkdir -p ~/.claude/skills
cp -r stancon-contributing/skills/stan-* ~/.claude/skills/
```

On Windows, in PowerShell:

```
New-Item -ItemType Directory -Force -Path "$HOME\.claude\skills"
Copy-Item -Recurse stancon-contributing\skills\stan-* "$HOME\.claude\skills\"
```

Then just describe what you are doing — no special syntax:

- *"I'd like to contribute to Stan but I don't know where to start"*
- *"I want to add a function to posterior — issue #238"*
- *"help me write the PR description"*

6.3.2 Without an agent

Nothing here needs an LLM:

- Read `SKILL.md` as a numbered recipe for the workflow.
- Copy `templates/bug-report.md` into a GitHub issue and fill in the comments.
- Run `Rscript .../check_environment_setup.R` from your clone before you push.
- Read `references/worked-example.md` to see a real contribution end to end.

The skills are the procedural companion to the checklist — same steps, more detail.

6.4 Mentor mode

The skills instruct the agent to be a mentor, not a contributor, see mentor-model.md.

- **You always know where you are.** Every code contribution passes through five stages — *orient, set up, change, verify, submit* — then review. Each workflow skill opens with that map and names the stage every time it moves.

  ```
  Orient    1. Issue read, unclaimed, and understood
  Set up    2. Fork and clone
            3. Check the environment
            4. Create a branch
  Change    5. Write the code   6. Document, then read your own diff
  Verify    7. Test   8. Update NEWS.md   9. Run devtools::check()
  Submit   10. Commit, push, open the pull request
  ```

  Only the contents differ between R, Python, and C++. The shape does not — so what you learn here transfers to the next repository.

- **The agents responds should adhere to the same structure:** The stage you are in, one sentence on why this step is necessary, one command block, and the offer *"Run it, or shall I?"*. Under 100 words, no recap, no preview.

- **Then it stops and waits.** *An unanswered offer is not permission.* This is the rule that does the work; without it an agent runs to the end alone.

Ask the agent to take a step over and it will, but it says what it did and what you must check, then hands the next step back. The one thing it should not take over is the understanding: you cannot delegate the review of code that goes out in your name.

The full statement is `stan-contributing/references/mentor-mode.md`

6.5 Which shell the commands assume

The command blocks in the skills — and in this book — are written for a POSIX shell: Terminal on macOS and Linux, and Git Bash on Windows. PowerShell and `cmd.exe` do not understand `2>/dev/null`, and neither ships `grep` or `head`.

Four commands genuinely differ on Windows, and the skills flag each one where it appears:

Written	On Windows
`source .venv/bin/activate`	`.venv\Scripts\Activate.ps1`
`python3`	`python`, or `py -3`
`./runTests.py …`	`python runTests.py …`
`make …` (Stan Math)	`mingw32-make …`, from RTools

Two toolchains have to be there before you start, on every platform. `devtools::check()` compiles the package, so it needs RTools on Windows and the Xcode command line tools (`xcode-select --install`) on macOS. Stan Math needs a C++ compiler, `make`, Python 3, and `doxygen` — all from RTools on Windows. Windows users, note also that the R installer does **not** put `Rscript` on your `PATH`.

The skills ask the agent to establish your platform at the set-up stage and then give you one command per turn, the one that fits your machine — not three variants to choose between.

6.6 A note on trusting them

The pull request goes out under your name, and you have to be able to explain every line of it. In the stan-dev repositories the AI Contribution Policy states this outright; the ArviZ repositories have no equivalent policy, but the same expectation holds in practice.

7 Walkthrough: Contributing to Stan's R packages

In this tutorial, we will walk you through one example code contribution to `posterior`, one of Stan's R packages. Along the way, we will show how you can use the Stan skills to support your contribution workflow.

Note that these skills are still a prototype. Please use them with care and report any problems you run into, so that we can learn how to improve them.

You can get the skills from https://github.com/florence-bockting/stancon-contributing. Fork and clone the repository. Open the repository of interest (in this case `posterior`) and save the stancon-contributing/skills folder under

- `~/.claude/skills` (should also be read by Cursor)
- `~/.codex/skills`
- `~/.copilot/skills`

depending on your LLM. Alternatively, use your LLM and prompt it:

```
read the skills from <path>/stancon-contributing/skills into your global skill
```

Even if you don't use an LLM, the skills are useful on their own: they provide recipes for the tasks that come up again and again when contributing to an R package.

7.1 Writing an issue

Suppose you hit a problem in the `posterior` package: all columns of the `summarise_draws()` output are converted to character, even the `mean` column, which should be numeric.

```
example_draws() |>
   summarise_draws(char_fun = \(x) "word", mean)

# A tibble: 10 × 3
#   variable char_fun mean
#   <chr>    <chr>    <chr>
# 1 mu       word     4.1799990610081
# 2 tau      word     4.16356885610418
# ...
```

First, we want to report this problem to the developers of the `posterior` package, so we write an issue. In Writing good Issues, we discussed what the structure of a *good* issue looks like.

The `stan-issue-and-pr` skill helps you write an issue for a given problem that follows this structure. The issue template is provided in skills/stan-issue-and-pr/templates/bug-report.md.

Next, we show how the skill can be used to create an issue.

7.1.1 Duplicate issue

Point the AI assistant at the skill and make sure it actually reads it.

You

```
I want to create an issue in stan-dev/posterior for the following problematic case:

example_draws() |>
   summarise_draws(char_fun = \(x) "word", mean)

Problem: all columns are converted to string even those that should be a double (e.g., mea
Help me write the Issue according to Stan conventions.
```

The first thing the LLM does is check for duplicates. You can look up all steps the skill asks for in skills/stan-issue-and-pr/SKILL.md.

In our case, an issue for this problem already exists, so the skill flags it as a duplicate.

Claude Opus Before drafting: I searched for duplicates first, and this exact case is already reported.

stan-dev/posterior#355 — "In summarise draws, a function that returns a character changes all columns to character", open since 2024-03-14, opened by n-kall, labelled bug. Its reproducible example is character-for-character your code:

```
example_draws() |>
   summarise_draws(char_fun = \(x) "word", mean)
```

The discussion there already diagnosed the cause: summarise_draws_helper binds results into a matrix and uses unlist, so a single character return coerces every column. Maintainers (mjskay, paul-buerkner) agreed the fix is to bind into a data frame directly rather than round-trip through a string.

There is also an open PR: #356 by n-kall, "modify summarise_draws internals not to coerce values to same type". It is stalled — last update 2025-03-03, using purrr::flatten, with a known failing test for functions returning an unnamed list or vector.

So a new issue would be a duplicate, and Stan maintainers close those. Two more useful paths:

- Comment on #355 if you have something new — a case the existing report misses, or evidence it still reproduces on 1.7.0.
- Pick up #356. It is unfinished, the maintainers want it, and the remaining problem is well-defined.

Which do you want to do? If it's the PR, I can help you draft the description with the stan-r-package-contribution workflow. If you think your case actually differs from #355, tell me how and we'll write the issue.

7.1.2 Drafting an issue

To see the rest of the workflow, we tell the LLM to skip the first step (i.e., the search for pre-existing issues).

You

```
Skip this step and help me draft an issue anyway as this is a demo.
```

The skills are written so that the AI assistant acts as a mentor: it guides you through the process step by step instead of working autonomously. This way, you learn the contribution process yourself and keep authority over every decision that is made.

Claude Draft is at issue-draft.md.

Three things are still yours to write. I left them marked rather than guessed:

- "What I expected" — I put a suggestion there. Rewrite it. Reviewers ask about text a contributor did not write.
- Additional context — did you hit this in real work or while reading code? Do you have a workaround?
- AI disclosure — the Stan AI Contribution Policy requires it in stan-dev repos, and you are literally on a branch adding that policy to CONTRIBUTING.md. Suggested wording is in the draft.

Also swap my environment block for your own sessionInfo().

issue-draft.md

```
# Title (suggestion - rewrite in your words)

summarise_draws() coerces all summary columns to character when one summary function retur

## What happened
```

````
`summarise_draws()` returns every summary column as `<chr>` if any one summary
function returns a character value. Numeric summaries such as `mean` become
strings.

## What I expected

<!-- SUGGESTION - this is your sentence to write. Mine below as a starting point. -->

Each summary column keeps its own type. `char_fun` is a character column and
`mean` stays a double. Only the column produced by the character-returning
function should be character.

## Minimal reproducible example

```r
library(posterior)

example_draws() |>
  summarise_draws(char_fun = \(x) "word", mean)
```

## Actual output

```
# A tibble: 10 × 3
   variable char_fun mean
   <chr>    <chr>    <chr>
 1 mu       word     4.1799990610081
 2 tau      word     4.16356885610418
 3 theta[1] word     6.74893947963962
 4 theta[2] word     5.25331634990726
 5 theta[3] word     3.04393475572924
 6 theta[4] word     4.85842854299997
 7 theta[5] word     3.22258991784476
 8 theta[6] word     3.98696993634695
 9 theta[7] word     6.5030995214442
10 theta[8] word     4.56520199862817
```

`str()` confirms the storage type, so this is not only a print-method effect:

```
drws_smm [10 × 3] (S3: draws_summary/tbl_df/tbl/data.frame)
 $ variable: chr [1:10] "mu" "tau" "theta[1]" "theta[2]" ...
 $ char_fun: chr [1:10] "word" "word" "word" "word" ...
````
````

```
 $ mean    : chr [1:10] "4.1799990610081" "4.16356885610418" ...
```

Steps to reproduce

1. `library(posterior)`
2. Run the example above.
3. Inspect the column types with `str()` or the tibble header.

Environment

- OS: Linux, x86_64-pc-linux-gnu
- Package version: posterior 1.7.0 (installed release)
- R version: 4.6.1 (2026-06-24)

<!-- Replace with your own sessionInfo() output before posting. -->

Additional context

<!-- OPEN - only you can fill these in:
     - Did you hit this in real work, or while reading the code?
     - Do you have a workaround you are using?
-->

Possible cause, from reading `R/summarise_draws.R`: `summarise_draws_helper()` pre-allocates a matrix at line 360 and fills it with `unlist()` at lines 362 and 367. A matrix holds one type, and `unlist()` on a mixed-type list coerces to character. Binding the per-variable results into a data frame instead of a matrix would avoid the round trip through character.

<!-- AI DISCLOSURE - required by the Stan AI Contribution Policy if it applies.
     Delete if you used no assistant. Suggested wording:
     "I used Claude Code to reproduce the example and locate the relevant code
     in summarise_draws.R. The report is in my own words." -->

Now suppose we want to contribute a fix for the issue above. The instructions for the contribution workflow are described in skills/stan-r-package-contribution/SKILL.md.

You can either work through the contribution yourself, using the skill as a guideline (or checklist), or you can work through it as a dialogue with an LLM.

7.2 Is the issue unclaimed?

You I want to contribute a solution to the issue: https://github.com/stan-dev/posterior/is-

sues/355

How do I get started?

Claude Here is the whole path. It is your checklist, not mine.

```
Orient    1. Issue read, unclaimed, and understood   ← you are here
Set up    2. Fork and clone
          3. Check the environment: roxygen2 pin, dev packages, clean tree
          4. Create a branch
Change    5. Write the code next to its siblings
          6. Document, then read your own diff
Verify    7. Test the failure modes, then the full suite
          8. Update NEWS.md
          9. Run devtools::check()
Submit   10. Commit, push, open the pull request
Review       Respond to reviewers
```

Stage: orient — step 1, confirm the issue is unclaimed.

Before any code, ask in the issue. Suggested comment — rewrite it in your words:

```
Is #356 still active? If not, I would like to pick this up. I plan to follow the data-
frame binding approach mjskay suggested rather than purrr::flatten, to avoid the new depen
vector case that PR hit.
```

Post that yourself, or shall I with gh?

While you wait, two questions decide the design. The thread contains a disagreement nobody resolved:

- purrr::flatten (n-kall's PR) or direct data-frame binding (mjskay, and paul-buerkner agreed)?
- PR #356 has a failing test for functions returning an unnamed list or vector. What should summarise_draws() do in that case?

Can you describe, in your own words, why the current code coerces the columns? If not, we read R/summarise_draws.R:340-375 together before step 2.

Skip this step and start with fresh local PR

7.3 Check setup

First, we ensure that we have a fork and clone of the repo of interest. There are different approaches for doing this, below are two different approaches using either `usethis` or `gh`.

```
# from R
usethis::create_from_github("stan-dev/posterior", fork = TRUE)

# from the shell, clones into ./posterior
gh repo fork stan-dev/posterior --clone
```

(In my case the LLM will notice that I have a clone of the package and push rights.)

Then, we check whether the environment is set up as expected. For this, the skills ship an R script, `stan-r-package-contribution/scripts/check_environment_setup.R`, that runs a few automatic checks.

Claude Two facts change the plan, both read-only:

You have push access to stan-dev/posterior ("push": true), so no fork is needed. Step 2 genuinely does not apply to you. Working tree is clean.

Stage: set up — step 3, check the environment.

This script reads your DESCRIPTION, your R library, and your git state. The check that matters is roxygen2: if your version differs from the one that documented this package, devtools::document() at step 6 turns a 4-file change into a 47-file diff of reformatted `\link{}` calls, and a reviewer cannot find your actual change in that. It changes nothing — it reads and prints one line per check.

```
Rscript /u/21/wa.bocktif1/unix/.claude/skills/stan-r-package-contribution/scripts/check_en
```

It will flag your current branch, add-ai-contribution-policy. That flag is correct — we cut the new branch from master in the next step, not from your policy work.

Run it, or shall I? Paste what it prints.

You

Copy & paste terminal output to LLM

```
user@posterior$ Rscript /u/21/wa.bocktif1/unix/.claude/skills/stan-r-package-contribution/

Environment check for an R package contribution.
Reads DESCRIPTION, your R library, and your git state. Changes nothing.
FAIL blocks a reviewable pull request. WARN is worth a look.

[OK  ] Package: posterior
```

```
[OK  ] Contributing guide: .github/CONTRIBUTING.md -- read it before you start.
[OK  ] roxygen2 7.3.3 matches the pin in DESCRIPTION.
[OK  ] devtools installed.
[OK  ] usethis installed.
[OK  ] testthat installed.
[OK  ] Working tree clean.
       Branch 'add-ai-contribution-policy' is not a base branch.
       Did you create it for this contribution? [y/N] N

[FAIL] Branch 'add-ai-contribution-policy' is not for this contribution.
       Your change would start from somebody else's work.
       git checkout main   # or master, or develop
       usethis::pr_init("123-short-description")
[OK  ] gh CLI authenticated.

Fix every FAIL above before you start the change.
```

Claude

Stage: set up — step 3, the FAIL on the branch.

```
Orient     1. Issue read, unclaimed, and understood
Set up     2. Fork and clone
           3. Check the environment  ← you are here
           4. Create a branch
Change     5. Write the code
           6. Document, then read your own diff
Verify     7. Test
           8. Update NEWS.md
           9. devtools::check()
Submit    10. Commit, push, open the pull request
```

The FAIL is correct. git log master..HEAD prints nothing, so add-ai-contribution-policy is already merged into master. New work here starts from a finished change.

This returns you to master and pulls the latest upstream commits, so your branch starts from current code:

```
usethis::pr_pause()
```

Run it, or shall I?

You Done.

Claude

Stage: set up — step 4, create a branch.

pr_init() pulls the latest upstream commits, creates the branch, and switches to it in one call. You are on master, so it starts from current code.

```
usethis::pr_init("355-summarise-draws-types")
```

The name appears in the pull request, so change the words if you prefer others.

Run it, or shall I?

Run the command and then we are on the respective development branch and can make our changes.

> **i** If `Rscript` is not found
>
> On Windows the R installer does not add `R\bin` to your `PATH`, so this command fails with "not recognized" before it checks anything. Either add `R\bin` to the `PATH`, or skip the script and run the same checks by hand from the R console — `packageVersion("roxygen2")` against the pin in `DESCRIPTION` is the one that matters. The script is a convenience; the checks are the point.
> One check further ahead has a platform requirement too: `devtools::check()` compiles the package, which needs RTools on Windows or the Xcode command line tools (`xcode-select --install`) on macOS. Sort that out here, at the set-up stage, rather than after the change is written.

7.4 Code contributions

The next step is to make the code changes. This step should always consist of two connected parts: (1) a test that checks the outcome you want to achieve, and (2) the code changes needed to achieve it.

In our example, we want `summarise_draws()` to return an output in which the type of one column is not carried over to the other columns.

So we could, for example, write the following tests into `tests/testthat/test-summarise_draws.R`.

```
test_that("summarise_draws_helper works correctly", {
  res1 <- example_draws() %>%
    summarise_draws(char_fun = function(x) "word", mean)

  expect_type(res1$char_fun, "character")
  expect_type(res1$mean, "double")

  res2 <- example_draws() %>%
```

```
    summarise_draws(char_fun = function(x) factor("a"), mean)

  expect_s3_class(res2$char_fun, "factor")
  expect_type(res2$mean, "double")

  res3 <- example_draws() %>%
    summarise_draws(char_fun = function(x) TRUE, mean)

  expect_type(res3$char_fun, "logical")
  expect_type(res3$mean, "double")
})
```

Then we make the code changes needed to reach the desired output. In this case, we change a helper function that is used by `summarise_draws()` in `R/summarise_draws.R`:

```
# flatten the summary list of a single variable into a list with one element
# per output column, keeping each value in its original type
flatten_summary_list <- function(v_summary) {
  unlist(lapply(v_summary, as.list), recursive = FALSE, use.names = FALSE)
}

summarise_draws_helper <- function(x, funs, .args) {
  variables_x <- variables(x)
  # get length and output names, calculated on the first variable
  out_1 <- create_summary_list(x, variables_x[1], funs, .args)
  the_names <- vector(mode = "list", length = length(funs))
  for (i in seq_along(out_1)){
    if (rlang::is_named(out_1[[i]])) {
      the_names[[i]] <- names(out_1[[i]])
    } else if (length(out_1[[i]]) > 1) {
      the_names[[i]] <- paste0(names(out_1)[i], ".", c(1:length(out_1[[i]])))
    } else {
      the_names[[i]] <- names(out_1)[i]
    }
  }
  the_names <- unlist(the_names)
  # Check for naming issues prior do doing lengthy computation
  if ("variable" %in% the_names) {
    stop_no_call("Name 'variable' is reserved in 'summarise_draws'.")
  }
  # Pre-allocate list to store output, one element per variable holding
  # the summary values of that variable as a flat (but untyped) list
  out <- vector("list", length(variables_x))
  out[[1]] <- flatten_summary_list(out_1)
  # Do the computation for all remaining variables
```

```
  if (length(variables_x) > 1L) {
    for (v_ind in 2:length(variables_x)) {
      out_v <- create_summary_list(x, variables_x[v_ind], funs, .args)
      out[[v_ind]] <- flatten_summary_list(out_v)
    }
  }
  # bind the summaries column-wise so that the type of each summary is
  # preserved instead of being coerced to a common type across summaries
  columns <- lapply(seq_along(the_names), function(i) {
    vctrs::vec_c(!!!lapply(out, .subset2, i))
  })
  names(columns) <- the_names
  out <- tibble::as_tibble(columns)
  out$variable <- variables_x
  out <- move_to_start(out, "variable")
  class(out) <- class_draws_summary()
  out
}
```

Finally, we run the tests to check whether our code changes make them pass.

```
devtools::load_all()

testthat::test_file("tests/testthat/test-summarise_draws.R")
```

7.5 Wrapping up

Once the code changes and tests are done, a few final tasks remain before we can open a pull request. For orientation, we can again check `skills/stan-r-package-contribution/SKILL.md` or simply ask the LLM:

You Changes are made. Is this branch PR ready?

Claude **Stage: verify — the branch is not PR ready yet.**

What is missing:

- Nothing is committed. git log master..HEAD is empty. Both files are still working-tree changes only.
- No NEWS.md entry (step 8). The file starts at # posterior 1.7.1, so you need a new # posterior (development version) header with a ### Bug Fixes bullet citing (#355).
- devtools::check() has not been run (step 9).
- No fork remote. origin is stan-dev/posterior itself. If you have push rights that is fine; otherwise usethis::pr_push() will fail.

Next step is NEWS.md. The skill is firm that you write this one — it is the only text users read under the package name.

Shall I show you the recent entries to match the style, or do you want to draft it straight away?

7.5.1 Update NEWS.md

The `NEWS.md` file keeps track of all changes made to the package over time. When a new release is published, it is the reference used to communicate which changes that release contains.

NEWS.md

```
# posterior (development version)

* Ensure `summarise_draws()` preserves the type of each summary, so that a
function returning a character or factor no longer coerces the numeric
columns to character (#355).

...
```

7.5.2 Run CRAN checks

Run the same checks that are run when a package is released to CRAN. This is a larger test suite that builds the vignettes and checks the documentation, the examples, and more. Depending on the package, this can take a while.

Claude

Then step 9. devtools::check() runs what CRAN and the CI run: it rebuilds the docs, runs the full suite in a clean session, and checks the examples and NEWS.md formatting. Finding a NOTE here costs you minutes; finding it in review costs you days. It takes several minutes.

```
devtools::check()
```

Run it, or shall I? Paste the summary lines it ends with.

You

Run `check()` command and paste output.

7.5.2.1 Add, commit, and push your changes (open your PR)

Next, add, commit, and push the changes with git. Commit in reasonable chunks and make sure that no unintended changes end up in the pull request. This is not always easy, especially for larger refactorings. Commit histories can be cleaned up afterwards (e.g., by squashing commits), but for the Stan R packages we would not recommend it: it is extra effort for the contributor without much benefit. Bottom line: try to work methodically and commit cleanly, but don't panic if your history is a bit messy. Just leave it as is.

```
git add R/summarise_draws.R
git commit -m "fix: summarise_draws_helper collects summary values column-wise"

git add tests/testthat/test-summarise_draws.R
git commit -m "tests: add tests for updated summarise_draws_helper"

git add NEWS.md
git commit -m "chore: update NEWS.md"
```

Finally, open a pull request with

```
usethis::pr_push()
```

7.6 PR description

The final task is to write the pull request description. The pr-description.md template helps you with this.

Following that template, a description for our example could look as follows:

pr-description.md

```
Closes #355

## Description

`summarise_draws()` collected all summary values in a matrix, which holds one
type, so a character-valued summary function coerced every column to character.
The values are now collected in a list per variable and bound column-wise with
`vctrs::vec_c()`, so each column keeps the type its function returned. This
follows the approach agreed in the issue thread.

## Breaking changes

A summary function whose return type differs across variables now raises an
```

```
error from `vec_c()` instead of coercing to character.

## Tests

Added a test that summary functions returning character, factor, and logical
leave the `mean` column `double`.

    [ FAIL 0 | WARN 0 | SKIP 11 | PASS 2032 ]
    devtools::check(): 0 errors | 0 warnings | 0 notes

## Documentation

`NEWS.md` bullet added. No user-facing docs changed; the new helper is internal.

## Open questions / Uncertainties

- I added the `skills/` folder to `.Rbuildignore`. If this is not desired, I can revert th

## AI assistance

Claude helped with drafting this PR. I reviewed every change and can
explain it.

## Copyright and Licensing

Copyright holder: USER or UNIVERSITY

By submitting this pull request, the copyright holder agrees to license the
submitted work under the following licenses:

- Code: BSD 3-clause (https://opensource.org/licenses/BSD-3-Clause)
- Documentation: CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/)

## Checklist

- [x] Style is consistent with the existing code and docs
- ~[ ] Documentation or vignette renders locally~
- [x] Tests pass, with the output pasted above
- [x] Full check passes (`devtools::check()` / `pytest` + `ruff`)
- [x] Changelog handled - `NEWS.md` entry added, or the title is
      changelog-worthy where the changelog is generated
- [x] Copyright holder named
- [x] Every file in the diff is explainable
- [x] AI assistance disclosed, per the
      [AI Contribution Policy](https://github.com/stan-dev/stan/wiki/AI-Contribution-Polic
```

8 Walkthrough: Contributing to ArviZ

In this tutorial, we walk you through one example code contribution to `arviz-stats`. A subpackage of ArviZ, a Python package that provides diagnostics and visualizations for the Bayesian workflow. This tutorial is the counterpart to Walkthrough: Contributing to Stan's R packages.

The contributing instructions for this workflow are described in `skills/stan-python-package-contribution/SKILL.md`, and the short version in form of a *checklist* is in Contributing to ArviZ (Python). The contributing instructions are based on the Contributing section in ArviZ.

In the following, we will show the contributing workflow using Stan skills. You can get the skills from https://github.com/florence-bockting/stancon-contributing. Fork and clone the repository. Open the repository of interest (in this case `posterior`) and save the stancon-contributing/skills folder under

- `~/.claude/skills` (should also be read by Cursor)
- `~/.codex/skills`
- `~/.copilot/skills`

depending on your LLM. Alternatively, use your LLM and prompt it:

```
read the skills from <path>/stancon-contributing/skills into your global skill
```

Even if you don't use an LLM, the skills are useful on their own: they provide recipes for the tasks that come up again and again when contributing to an R package.

You can work through the contribution yourself, using the skill as a checklist, or work through it as a dialogue with an LLM, as we do below.

8.1 Example

The change: add the Brier score to `metrics()`, mirroring `measure_brier` on loo's `pred_measure` branch.

`metrics()` already computes accuracy for a binary outcome. The goal is that `kind="brier"` works the same way:

```
from arviz_base import load_arviz_data
from arviz_stats import metrics

dt = load_arviz_data("anes")
metrics(dt, kind="acc")     # works today
metrics(dt, kind="brier")   # what we want to add
```

8.2 Claim the issue, and check the feature does not already exist

ArviZ has explicit etiquette here, and it comes *before* any code:

- **comment on the issue before starting work**
- if the issue is already assigned, ask the assignee first
- the comment is yours to write and post. An LLM can help you draft it; do not let it post for you

8.3 Create the issue

We will create an issue with assistance from an LLM:

You Read the skills in stancon-contributing/skills I want to add a function to Arviz-stats and for this I want to write first an Issue. Support me drafting it.

Here is the idea: add the Brier score to `metrics()`, mirroring `measure_brier` on loo's `pred_measure` branch.

`metrics()` already computes accuracy for a binary outcome. The goal is that `kind="brier"` works the same way:

```
from arviz_base import load_arviz_data
from arviz_stats import metrics

dt = load_arviz_data("anes")
metrics(dt, kind="acc")     # works today
metrics(dt, kind="brier")   # what we want to add
```

After some clarification questions…

- What is the problem?
- What is the current approach?
- Is there a related PR?
- What is the scope of the PR (small, medium, large)?
- Do I want to work on the problem?

… the LLM provides a draft for a **feature request** which needs to be checked by the user.

issue-draft.md

```
# Add Brier score to metrics()

## The problem

`metrics()` cannot compute the Brier score.

`metrics()` already supports binary outcomes through `kind="acc"` and
`kind="acc_balanced"`. Those score a hard classification. The Brier score
scores the predicted probabilities instead, so it is the natural companion
measure and its absence is inconsistent.

## What I currently do instead

I compute it by hand from the posterior predictive draws.

## Proposed solution (optional)

Add `kind="brier"` so it works the same way as the existing binary kinds:

    from arviz_base import load_arviz_data
    from arviz_stats import metrics

    dt = load_arviz_data("anes")
    metrics(dt, kind="acc")     # works today
    metrics(dt, kind="brier")   # what this issue asks for

This mirrors `measure_brier` in the loo package.

## Scope

- [x] Small and self-contained - one function, no API change
- [ ] Medium - touches several functions or adds an argument to a public API
- [ ] Opens a design discussion

## Related

- `measure_brier` in stan-dev/loo#363 ("Adding `pred_measure` feature",
  draft, branch `pred_measure`).
- Existing binary kinds in `metrics()`: `acc`, `acc_balanced`.

## Would you be willing to work on this?
```

```
Yes.
```

8.4 Setup environment

ArviZ has a modular stack consisting of three modules:

Repository	Holds	Your change goes here if…
`arviz-base`	data structures, `DataTree`/`InferenceData`, converters	you are changing how data is *represented*
`arviz-stats`	statistics, diagnostics, LOO, metrics	you are computing a *number*
`arviz-plots`	plots, multi-backend rendering	you are drawing something

This mirrors the "which stan-dev repository?" question from The Stan ecosystem. A new metric is a number, so: `arviz-stats`.

You I want to contribute to Arviz. How do I get started?

Claude Opus

Here is the map for a Python contribution:

```
Orient     1. Pick the right repository of the three
           2. Comment on the issue to claim it; check the feature does not exist
Set up     3. Fork, clone, add the upstream remote
           4. Install: tox, plus an editable install
           5. Install the pre-commit hooks
           6. Branch (main is protected by a hook)
Change     7. Write the code at the right architectural layer
           8. numpydoc docstrings; update every duplicated list
Verify     9. Parametrised pytest tests with computed expected values
          10. Check no test skips for a missing dependency; run tox
Submit    11. Rebase on upstream/main, push
          12. Do NOT edit CHANGELOG.md; write a changelog-worthy PR title
Review        Respond to reviewers  →  stan-issue-and-pr
```

Stage: Orient — step 1, pick the repository.

ArviZ is three packages now. Which one you target depends on what your change produces, so we settle that before anything else.

Every code contribution runs through the same five stages:

`Orient → Set up → Change → Verify → Submit → Review`

Now, we would declare which Issue we want to work on in which repository. In the current example this is `arviz-stats`.

Claude Opus **Stage: Set up — step 3, fork and clone.**

origin becomes your fork. upstream becomes the real project. Step 11 rebases onto upstream/main, so both remotes must exist from the start.

You are currently in `<some branch>`, so pick a directory outside it first.

```
gh repo fork arviz-devs/arviz-stats --clone
cd arviz-stats
git remote add upstream git@github.com:arviz-devs/arviz-stats.git
```

Create fork and clone it. Setup the remote.

Claude Opus **Stage: set up — step 4, install.**

First, a read-only look, so we do not build a second environment on top of one you already have:

```
ls -d .venv env 2>/dev/null; python3 -c "import arviz_stats, sys; print(arviz_stats.__file
```

Shall I run this, or will you?

You I got: `ModuleNotFoundError: No module named 'arviz_stats'`

Claude Opus **Stage: set up — step 4, install.**

The editable install makes import arviz_stats read the file you edit, not a copy from PyPI. Without it your changes do nothing.

```
python3 -m venv .venv
source .venv/bin/activate
pip install tox
pip install -e ".[test]"
```

arviz-stats needs only numpy and scipy, so this is quick. The [test] extra adds pytest and the fixtures; without it the test files do not import.

Run it, or shall I?

> **i** This transcript was recorded on Linux
>
> The two lines above are the only ones in this walkthrough that differ by platform. On Windows the interpreter is `python` (or `py -3`), and `venv` writes `Scripts\` where macOS and Linux write `bin/`:

```
python -m venv .venv
.venv\Scripts\Activate.ps1
```

In Git Bash it is `source .venv/Scripts/activate`. Everything after the activation line — `pip`, `tox`, `pytest`, `pre-commit`, `git` — is identical on all three platforms. The skill asks the agent to establish your platform at this step and give you the matching form; see Which shell the commands assume.

You Ok. All installed.

Claude Opus **Stage: set up — step 4, confirm the install.**

An editable install points at your working copy. If it points into site-packages, your edits are invisible and step 7 wastes an hour:

```
.venv/bin/python -c "import arviz_stats; print(arviz_stats.__file__)"
```

Shall I run this?

You Returned: /u/21/wa.bocktif1/unix/GitHub/arviz-stats/src/arviz_stats/**init**.py

Claude Opus That path is your clone, so edits take effect immediately.

Stage: set up — step 5, install the pre-commit hooks.

ArviZ enforces style with ruff and pylint at commit time. The hooks fix formatting on your machine instead of failing in public CI:

```
.venv/bin/pip install pre-commit pylint && .venv/bin/pre-commit install
```

pylint is installed here as well, because its hook is declared `language: system`. Pre-commit does not build an environment for that hook. It calls pylint on the PATH, and fails if the package is absent. Every other hook, ruff included, brings its own pinned copy.

Run it, or shall I?

You done

Claude Opus **Stage: set up — step 5, see what the hooks do.**

Run them once now, on a clean tree, so you learn what they change before one rewrites a file you are mid-way through editing:

```
.venv/bin/pre-commit run --all-files
```

The first run downloads the pinned tool versions, so it is slow. Run it, or shall I?

You done

Claude Opus **Stage: set up — step 6, branch.**

Cut it from a main that matches upstream, or you inherit conflicts at step 11:

```
git checkout main
git pull upstream main
git checkout -b add-brier-score
```

Run it, or shall I?

8.5 Code contribution

Steps 7 and 8 in the contribution workflow refer to making the code contribution and adding function documentation using docstrings.

R packages are flat: one file in `R/`, one entry in `NAMESPACE`. arviz-stats is layered, and a new metric touches each layer in turn. Docstrings follow **numpydoc**.

The computation goes in the array layer, next to `_mae`, `_mse` and `_acc`:

We need additional validation and documentation in the public layer:

Furthermore, we need to update the docstring at two places. `valid_kind` is validated in one place, but the list of options is written out again in the docstring of the public `metrics()` *and* in the docstring of the private `_metrics()`.

8.6 Add test with `pytest`

Tests are parametrised, using a shared fixture:

You Changed code, docstring, and added test.

Claude Opus

Stage: verify — step 10, run the tests.

A missing dependency skips a test instead of failing it. -rs prints one line per skip, so you see whether your new test actually ran.

```
pytest tests/test_metrics.py -q -rs
```

Run it, or shall I?

You (.venv) wa.bocktif1@l26-0069:~/GitHub/arviz-stats$ pytest tests/test_metrics.py -q -rs ……. [100%] 44 passed in 3.80s

Listing 8.1 `src/arviz_stats/base/diagnostics.py`

```
@staticmethod
def _brier(observed, predicted):
    """Compute the Brier score.

    Parameters
    ----------
    observed: array-like of shape = (n_outputs,)
        Ground truth (correct) target values, coded as 0 or 1.
    predicted: array-like of shape = (n_outputs)
        Predicted probabilities of the outcome being 1.

    Returns
    -------
    mean: float
        Brier score. Lower values indicate better calibrated predictions.
    std_error: float
        Standard error of the Brier score.
    """
    n_obs = len(observed)
    sq_e = (predicted - observed) ** 2
    mean = np.mean(sq_e)
    std_error = np.std(sq_e) / n_obs**0.5
    return mean, std_error
```

Add-on: Debugging (click to expand)

In case you need to debug something in your Python code you can do the following:

- Go to the function that you want to debug
- Insert a `breakpoint()` in the line from where you want to start looking-up the code

Example:

Listing 8.2 src/arviz_stats/metrics.py

```
-    valid_kind = ["mae", "rmse", "mse", "acc", "acc_balanced"]
+    valid_kind = ["mae", "rmse", "mse", "acc", "acc_balanced", "brier"]
```

Listing 8.3 src/arviz_stats/metrics.py

```
def metrics(data, kind="rmse", var_name=None, sample_dims=None, round_to=None):
    """
    Compute performace metrics.

    Currently supported metrics are mean absolute error, mean squared error and
    root mean squared error.
    For classification problems, accuracy and balanced accuracy are also supported.

    Parameters
    ----------
    data: DataTree or InferenceData
        It should contain groups `observed_data` and `posterior_predictive`.
    kind: str
        The kind of metric to compute. Available options are:

        - 'mae': mean absolute error.
        - 'mse': mean squared error.
        - 'rmse': root mean squared error. Default.
        - 'acc': classification accuracy.
        - 'acc_balanced': balanced classification accuracy.
>>      - 'brier': Brier score for binary classification.
...
```

```
    @staticmethod
    def _brier(observed, predicted):
        """Compute the Brier score.

        Parameters
        ----------
        observed: array-like of shape = (n_outputs,)
            Ground truth (correct) target values, coded as 0 or 1.
        predicted: array-like of shape = (n_outputs)
            Predicted probabilities of the outcome being 1.

        Returns
        -------
        mean: float
            Brier score. Lower values indicate better calibrated predictions.
        std_error: float
            Standard error of the Brier score.
        """
        n_obs = len(observed)
>>      breakpoint()
        sq_e = (predicted - observed) ** 2
        mean = np.mean(sq_e)
        std_error = np.std(sq_e) / n_obs**0.5
        return mean, std_error
```

Listing 8.4 `src/arviz_stats/metrics.py`

```
def _metrics(observed, predicted, kind, round_to):
    """Compute performance metrics.

    Parameters
    ----------
    observed: DataArray
        Observed data.
    predicted: DataArray
        Predicted data.
    kind: str
        The kind of metric to compute. Available options are:

        - 'mae': mean absolute error.
        - 'mse': mean squared error.
        - 'rmse': root mean squared error. Default.
        - 'acc': classification accuracy.
        - 'acc_balanced': balanced classification accuracy.
>>      - 'brier': Brier score for binary classification.
...
```

- Then run the test with the `--pdb` flag
- Use for example `n(ext)` or `s(tep)` to navigate through your code and evaluate variables (see Python Debugger)

Listing 8.5 `tests/test_metrics.py`

```
@pytest.mark.parametrize(
    "kind, round_to, expected_mean, expected_se",
    [
        ("acc", 2, 0.43, 0.19),
        ("acc_balanced", "2g", 0.46, 0.039),
>>      ("brier", "2g", 0.26, 0.0094),
    ],
)
def test_metrics_acc(datatree_binary, kind, round_to, expected_mean, expected_se):
```

```
(.venv) wa.bocktif1@l26-0069:~/GitHub/arviz-stats$ pytest tests/test_metrics.py --pdb
========================================== test session starts ==========================
platform linux -- Python 3.12.3, pytest-9.1.1, pluggy-1.6.0
rootdir: /u/21/wa.bocktif1/unix/GitHub/arviz-stats
configfile: pyproject.toml
plugins: cov-7.1.0
collected 44 items

tests/test_metrics.py ..........................
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PDB set_trace (IO-capturing turned off) >>>>>>>>>>>>>>>>
> /u/21/wa.bocktif1/unix/GitHub/arviz-stats/src/arviz_stats/base/diagnostics.py(1746)_br
-> sq_e = (predicted - observed) ** 2
(Pdb) n_obs
7
(Pdb) n
> /u/21/wa.bocktif1/unix/GitHub/arviz-stats/src/arviz_stats/base/diagnostics.py(1747)_br
-> mean = np.mean(sq_e)
(Pdb) n
> /u/21/wa.bocktif1/unix/GitHub/arviz-stats/src/arviz_stats/base/diagnostics.py(1748)_br
-> std_error = np.std(sq_e) / n_obs**0.5
(Pdb) mean
<xarray.DataArray 'y' ()> Size: 8B
array(0.25816518)
(Pdb)
```

- quit the debug mode with `q`

Claude Opus **Stage: verify — step 10, run the full environment.**

44 passed, no skips. Good. tox -e full sets ARVIZ_REQUIRE_ALL_DEPS=TRUE, so a skip becomes an error there. It is close to what CI runs. The first run is slow.

```
tox -e full
```

Run it, or shall I?

You Ends with

```
================================== 2727 passed, 2 skipped in 109.59s (0:01:49) ============
build: _exit> python /u/21/wa.bocktif1/unix/GitHub/arviz-stats/.venv/lib/python3.12/site-
packages/pyproject_api/_backend.py True flit_core.buildapi
  full: OK (116.82=setup[3.13]+cmd[113.69] seconds)
  congratulations :) (116.87 seconds)
```

Claude Opus **Stage: verify — step 10, last part.**

2727 passed, 2 skipped.

Style is enforced by pre-commit here, so a lint failure blocks the commit. This runs the same checks now:

```
tox -e check
```

Run it, or shall I?

First time failed for *end of files*.

```
check for added large files..............................................Passed
check toml...............................................................Passed
check for merge conflicts................................................Passed
fix end of files.........................................................Failed
- hook id: end-of-file-fixer
- exit code: 1
- files were modified by this hook

Fixing src/arviz_stats/base/diagnostics.py

don't commit to branch...................................................Passed
trim trailing whitespace.................................................Passed
ruff check...............................................................Passed
ruff format..............................................................Passed
Disallow print statements................................................Passed
pylint...................................................................Passed
pre-commit hook(s) made changes.
```

This is fixed directly, so just run command again:

```
(.venv) wa.bocktif1@l26-0069:~/GitHub/arviz-stats$ tox -e check
check: commands[0]> pre-commit install
pre-commit installed at .git/hooks/pre-commit
check: commands[1]> pre-commit run --all-files --show-diff-on-failure
check for added large files..............................................Passed
check toml...............................................................Passed
check for merge conflicts................................................Passed
fix end of files.........................................................Passed
don't commit to branch...................................................Passed
trim trailing whitespace.................................................Passed
ruff check...............................................................Passed
ruff format..............................................................Passed
Disallow print statements................................................Passed
pylint...................................................................Passed
```

8.7 Wrapping up

8.7.1 Git add, commit, rebase, and push

Now, we can go ahead and add and commit the changes via git. Try to keep your commit history clean and commit not all files at once but in reasonable conceptual chunks that let a reviewer infer your workflow.

Unlike the R packages, ArviZ asks you to **rebase onto `upstream/main` before pushing**. This puts your commits on top of the current upstream, so the pull request shows your change and nothing else. It is why we added the `upstream` remote back in step 3.

Claude Opus **Stage: submit — step 11, commit, rebase, and push.**

check passed. add names exactly the files that belong to this change, so nothing unrelated joins the commit. fetch downloads the new upstream work but changes nothing locally. rebase puts your commits on top of it. push sends the branch to your fork, and the work becomes public in your name.

```
git add src/arviz_stats/base/diagnostics.py src/arviz_stats/metrics.py tests/test_metrics.
git commit -m "Add Brier score to metrics()"
git fetch upstream
git rebase upstream/main
git push -u origin add-brier-score
```

One at a time, not as a block. If the rebase reports conflicts, stop and we work through them together. If the change is not finished, mark the pull request `[WIP]` — that is the project's convention.

Run them, or shall I?

You Skip this step as this is a demo. But support with creating a PR description.

> ℹ Different to R-packages: do **not** edit the changelog
>
> `CHANGELOG.md` in arviz-stats is generated at release time from merged pull request titles.
> There is no "unreleased" section to append to. Editing it by hand creates a conflict at the next release.
> **Your PR title becomes the changelog entry**, so it carries the weight that `NEWS.md` carries in the R packages. Write it accordingly:
>
> - Good: `Add Brier score to metrics()`
> - Avoid: `update metrics`

Claude Opus **Stage: submit — step 12, the description.**

arviz-stats has no pull request template, so we use the standard sections. CHANGELOG.md is generated from titles here, so do not edit it.

The description must be your words. I ask, you answer, then we assemble.

```
1. In one sentence of your own: what can a user do now that they could not do before?
2. Is there a decision a reviewer could question?
3. Did an AI assistant help you? Name what it did. ArviZ has no policy requiring
this, but it is good practice and reviewers appreciate knowing.
4. Who is the copyright holder? You, or your employer or university, if they own this work
```

You

```
1. `metrics()` supports now also the brier score.
2. Not that I know.
3. Yes, Claude Code assisted me. But I went over the changes myself.
3. Aalto university
```

pr-description.md

```
Title: Add Brier score to metrics()

Closes #<ISSUE NUMBER>

## Description

`metrics()` now supports the Brier score. `_brier()` sits next to `_mae` and
`_mse` in `base/diagnostics.py` and returns `(mean, std_error)` like they do.
`"brier"` is added to `valid_kind` in `_metrics()` and to the `kind` list in
```

```
both docstrings.

## Breaking changes

None.

## Tests

`test_metrics_acc` gains a `"brier"` case, with the expected mean and standard
error computed from the `datatree_binary` fixture.

`tox -e full`:

    2727 passed, 2 skipped

`tox -e check`:

    check: OK

## Documentation

numpydoc docstring on `_brier()`, plus the `'brier'` entry in the `kind` list
of `metrics()` and `_metrics()`.

## Open questions

- None

## AI assistance

Claude Code assisted with this change. I reviewed every line and can explain it.

## Copyright and Licensing

Copyright holder: Aalto University

By submitting this pull request, the copyright holder agrees to license the
submitted work under the repository's license, Apache License 2.0.
```