StatsClaw: An AI-Collaborative Workflow for Statistical Software Development*

Tianzhu Qin (Cambridge)

Yiqing Xu (Stanford)

April 4, 2026

Abstract

Translating statistical methods into reliable software is a persistent bottleneck in quantitative research. Existing AI code-generation tools produce code quickly but cannot guarantee faithful implementation—a critical requirement for statistical software. We introduce StatsClaw, a multi-agent architecture for Claude Code that enforces information barriers between code generation and validation. A planning agent produces independent specifications for implementation, simulation, and testing, dispatching them to separate agents that cannot see each other's instructions: the builder implements without knowing the ground-truth parameters, the simulator generates data without knowing the algorithm, and the tester validates using deterministic criteria. We describe the approach, demonstrate it end-to-end on a probit estimation package, and evaluate it across three applications to the authors' own R and Python packages. The results show that structured AI-assisted workflows can absorb the engineering overhead of the software lifecycle while preserving researcher control over every substantive methodological decision.

Keywords: statistical software, AI-assisted workflows, verification, Monte Carlo diagnostics, causal inference

*Tianzhu Qin, PhD Candidate, The Centre for Human-Inspired Artificial Intelligence, University of Cambridge. Email: tq224@cam.ac.uk. Yiqing Xu, Assistant Professor, Department of Political Science, Stanford University. Email: yiqingxu@stanford.edu. The authors used Claude Code as research, coding, and writing assistants in preparing this manuscript. All interpretations, conclusions, and any errors remain solely the responsibility of the authors. We thank Euphie Wang for designing the StatsClaw mascot.

1. Introduction

Methodological innovation in statistics and the social sciences begins with theoretical analysis but ultimately depends on software. A new estimator reaches practitioners through a package—an R library on CRAN, a Python module on PyPI, a Stata command on SSC. Producing that package requires translating mathematical derivations into algorithms, implementing those algorithms in code, writing tests that verify numerical correctness, building documentation that explains usage, and maintaining all of these as the package evolves. For most methodological teams, this engineering work is the binding constraint (Ram et al., 2019).

The scale of the problem has grown. Modern causal inference packages support multiple estimators, cross-validation procedures, bootstrap inference, C++ backends, and visualization systems. Keeping code, documentation, and tests mutually consistent across these components is labor-intensive and error-prone. When a research team modifies a scoring function, the change propagates through parameter interfaces, bootstrap routines, plotting code, and user-facing documentation. Missing a single downstream dependency produces bugs that may be silent—numerically plausible but statistically incorrect (McCullough and Vinod, 1999).

These costs have consequences. Methods that lack reliable software are adopted slowly, if at all. Packages that ship without adequate testing may produce incorrect results in edge cases. Documentation that falls out of sync with code misleads users about what the software actually computes. The result is a gap between the methods that exist in the literature and the methods that are available in practice (Peng, 2011).

Recent advances in AI code generation have begun to narrow this gap. Large language models can now produce syntactically valid code at impressive speed, from function-level generation (Chen et al., 2021) to repository-level issue resolution (Jimenez et al., 2024) and agentic interaction with codebases (Yang et al., 2024). But for statistical software, speed

of generation is not the binding constraint—correctness is. The standard workflow for AI-assisted development is *generate, then test*: the same model (or the same information set) produces both the code and the tests. This creates a structural problem. If the generator misunderstands the specification—conflating the observed and expected information matrix, using the wrong covariance estimator variant, mishandling truncation in MCMC sampling—it will likely embed the same misunderstanding in its tests. The code passes, but the implementation is wrong.

This paper introduces StatsClaw, an AI-assisted workflow designed around a simple principle: *the process that generates code must never be the same process that validates it.* Concretely, StatsClaw is a set of agent prompts and configuration files for Anthropic's Claude Code—there is nothing to install beyond Claude Code itself, and the "agents" are dispatched within a single session rather than running as separate processes. The framework's value lies entirely in the *structure* it imposes: information barriers between code generation and code validation, mandatory comprehension checks before any code is written, and a state machine that enforces sequential gates. A planning agent reads the source material (mathematical derivations, existing codebases, pseudocode, algorithm descriptions) and produces two independent documents: a *specification* telling a builder agent what to implement, and a *test specification* telling a tester agent what to verify. Neither agent sees the other's document. When the workflow includes simulation-based diagnostics, a third document drives an independent Monte Carlo evaluation that treats the implementation as a black box.

We describe the approach (Section 2), walk through a complete example from source material to installable R package (Section 3), and evaluate the workflow across three applications of increasing complexity (Section 4): paper-to-feature development, code translation, and sustained multi-day refactoring. Section 5 discusses limitations, scope, and implications. Appendix A provides the full architecture reference and a practical guide for adoption.

2. The Approach

The central problem in AI-assisted statistical software development is not code generation but *verification*. A language model that misunderstands a mathematical specification will produce code that is syntactically valid, passes superficial checks, and silently computes the wrong thing. The standard generate-then-test workflow amplifies this risk: the same information that produced the misunderstanding also produces the tests, so errors correlate rather than cancel. StatsClaw addresses this by structuring AI-assisted development around specialized agents, enforced information barriers, and mandatory comprehension checks.

2.1. Agents and Orchestration

StatsClaw orchestrates eight specialized agents[1] within a single Claude Code session (Figure 1). Each agent has a defined role, a restricted information set, and a fixed position in the workflow's state machine. The system is implemented as a set of agent prompts and configuration files for Claude Code; the "agents" are dispatched via Claude Code's built-in Agent tool within a single session, not as separate processes or services.

Leader is the orchestrator. It receives the user's request, selects the appropriate workflow type (from ten supported patterns), dispatches all other agents, and enforces a state machine with mandatory preconditions at each transition. It is the only agent that communicates directly with the user. Not every task requires the full pipeline: a simple bug fix may need only the builder and tester, while a greenfield package invokes the planner, all three execution agents, and the full review chain. The leader makes this decision, which is why it is a role distinct from the planner. **Planner** is the bridge between the user's source material and the execution pipelines. It reads all inputs—mathematical derivations, existing

[1]The current implementation includes a ninth agent, the *distiller*, which supports an optional shared-knowledge system ("Brain mode") currently under development. The distiller extracts reusable, privacy-scrubbed insights from completed workflow runs and, with explicit user consent, contributes them to a shared knowledge repository. Brain mode is planned for a future release and is not discussed in this paper. When it is disabled—the default—the eight-agent architecture described here is complete.

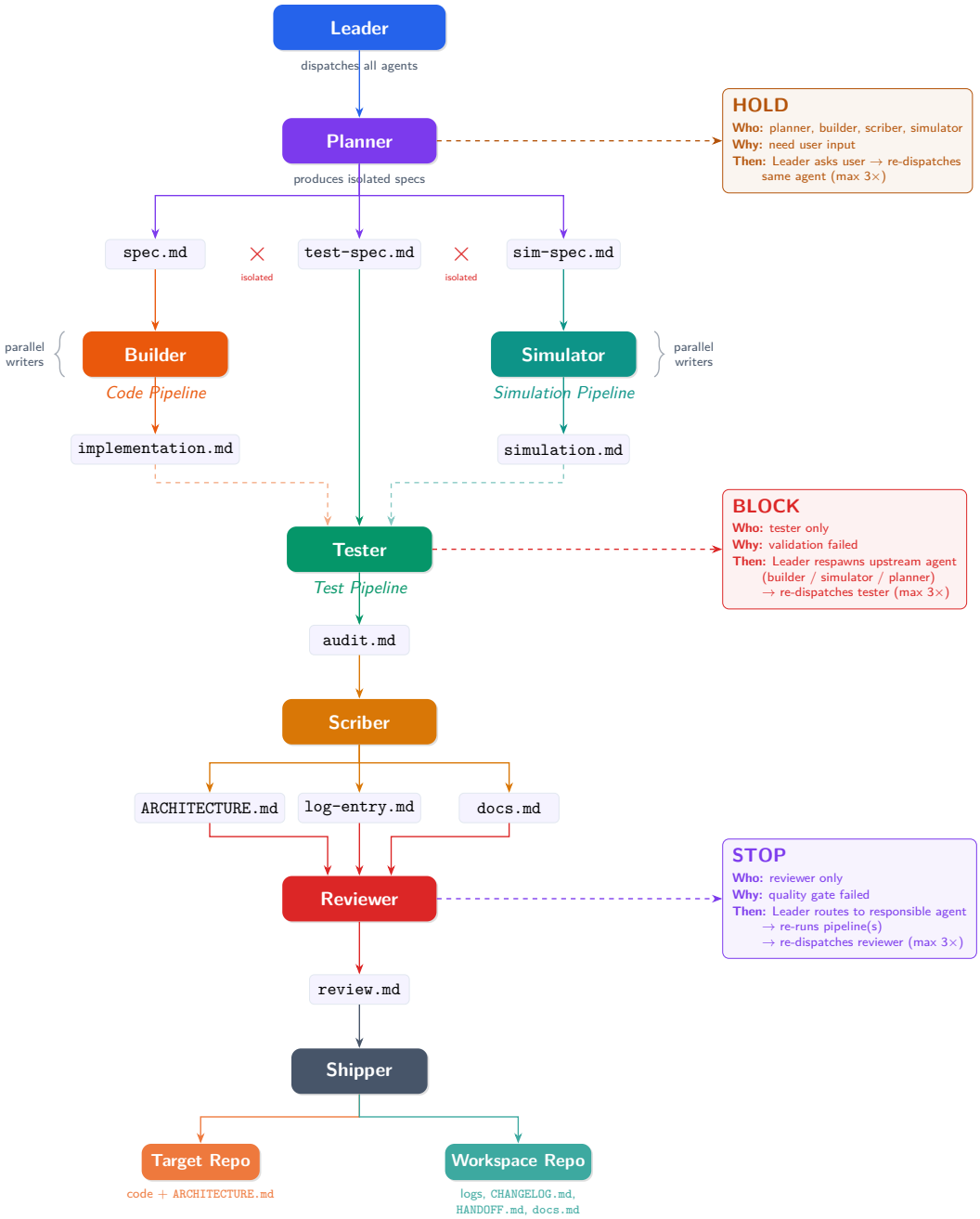

Figure 1. StatsClaw workflow architecture. The planner produces three isolated specification documents; the builder, tester, and simulator each receive only their own specification (× marks information barriers). The reviewer cross-compares all pipeline outputs before issuing a ship verdict.

codebases, pseudocode, algorithm descriptions—and produces the specification documents that downstream agents will consume. The planner is the only agent with access to the full information set; every other agent sees only the slice it needs.

Three agents form the execution layer, each operating on its own specification in an isolated git worktree. **Builder** implements the software: writing source code, configuring build systems, and producing an installable package. **Simulator** designs and runs simulation experiments that treat the implementation as a black box. **Tester** independently constructs a validation suite: unit tests, edge cases, property-based checks, and cross-reference benchmarks against established implementations. The tester's acceptance criteria are hard-coded and version-controlled: matching ground-truth parameters within a specified tolerance is a deterministic check, not a judgment call by the language model. These three agents execute in parallel and cannot communicate with one another.

Scriber documents the process after the execution pipelines complete, recording architecture decisions, implementation details, and a structured log of what each agent did. Because the scriber operates with a long accumulated context, it risks *rule drift*—forgetting constraints imposed earlier in the session. **Reviewer** mitigates this by receiving only the scriber's condensed output, enforcing rules from a fresh context window. The reviewer is the convergence gate: it is the only agent besides the planner with cross-pipeline visibility, reading all specification documents, all pipeline outputs, and the scriber's documentation, then auditing whether the pipelines converge—whether the builder's implementation satisfies the tester's behavioral contracts and the simulator's Monte Carlo criteria. The reviewer issues a ship or no-ship verdict. Finally, **Shipper** handles git operations (commit, push, pull request) upon user authorization.

Three interrupt signals coordinate the workflow when things go wrong. A HOLD signal, which any agent can raise, requests user input; the leader forwards the question and re-dispatches upon receiving an answer. A BLOCK signal, which only the tester can raise, indicates a validation failure; the leader re-dispatches the builder with the failure details, and the tester re-validates after the fix. A STOP signal, which only the reviewer can raise, indicates a quality-gate failure and routes back to the responsible agent. Each signal permits a maximum of three retry cycles before escalating to the user. The state machine

progresses through nine states—from credential verification through planning, specification, pipeline execution, documentation, review, and shipping—with hard gates at each transition. A detailed description of each agent's responsibilities, the full state machine, and the ten supported workflow types is provided in Appendix A.

2.2. Information Barriers

The architecture described above would offer little advantage over a single-agent workflow if all agents shared the same information. The key design constraint is what each agent *cannot* see.

In the full workflow, the planner produces three self-contained specification documents from the source material. The implementation specification (`spec.md`) describes the algorithm's computational steps, data structures, input validation rules, and API contracts; it is given to the builder. The simulation specification (`sim-spec.md`) specifies the data-generating process, scenario grid, performance metrics, and acceptance criteria; it is given to the simulator. The test specification (`test-spec.md`) describes expected behaviors, numerical tolerances, edge cases, and property-based invariants; it is given to the tester. Each document is independently sufficient for its recipient. Neither references the other. At dispatch time, the builder receives only the implementation specification and never sees the test specification; the tester receives only the test specification and never sees the implementation specification or the source code. Table 1 summarizes the full access matrix.

Table 1. Information barriers: each agent sees only its own specification.

Agent	Model Spec.	Simulation Spec.	Test Spec.
Builder	✓	×	×
Simulator	×	✓	×
Tester	×	×	✓

This separation prevents each agent from *teaching to the test*—finding surface-level solutions that pass validation without genuinely satisfying the specification. The failure modes

are concrete and predictable. If the builder knows the ground-truth parameters used for validation, it can hardcode those values rather than implementing the algorithm correctly—the tests pass, but the implementation is wrong. If the simulator knows how the algorithm works, it can design trivially easy data-generating processes that the implementation passes without being tested on difficult cases. If the tester sees the source code, it may design tests that follow the implementation's logic rather than independently verifying behavioral contracts. By enforcing information barriers, StatsClaw ensures that the builder must implement the algorithm from its specification alone, the simulator must design DGPs from the mathematical theory alone, and the tester must validate purely by checking whether ground truth is recovered—a deterministic comparison against the planner's criteria. A bug that survives must simultaneously satisfy two or three independently derived behavioral contracts, analogous to independent replication in experimental science.

2.3. Deep Comprehension as a Hard Gate

Information barriers ensure that downstream agents cannot contaminate each other's work, but they do nothing to prevent the planner itself from misunderstanding the source material. If the planner misreads a formula or conflates two estimator variants, it will embed that misunderstanding in all three specification documents, and the information barriers will faithfully propagate a coherent but wrong set of instructions. StatsClaw addresses this through a mandatory comprehension protocol that serves as a hard gate before any specification is written.

The protocol requires the planner to inventory every equation, symbol, assumption, and theorem from the source material; restate all mathematical content in its own notation, defining every symbol's type, dimensions, and domain; self-test against diagnostic questions (can the core requirement be restated without reference to the source? can every formula be reproduced and explained? have implicit assumptions been identified?); and produce an auditable `comprehension.md` artifact that the user can review before any code is generated.

This artifact is the single point of human oversight in the workflow: if the planner's understanding is correct, the downstream specifications will be correct; if it is wrong, the user catches it here rather than debugging silent numerical errors downstream.

If the protocol yields only partial understanding, the agent raises a HOLD signal—a structured request for user clarification. The system does not proceed until comprehension is verified. This shifts the failure mode from "generated wrong code" to "asked the right clarifying questions."

3. Demonstration: Implementing Probit Regressions

To make the workflow concrete, we walk through a complete application: building an R package implementing three probit estimation methods from a short PDF specification, with Monte Carlo simulation to evaluate finite-sample performance. This example uses the three-pipeline architecture (builder, tester, and simulator) and illustrates every stage of the workflow.

3.1. The Problem

The probit model is a standard binary choice model. The outcome is generated through a latent variable $y_i^* = \mathbf{x}_i'\boldsymbol{\beta} + \varepsilon_i$, where $\varepsilon_i \sim \mathcal{N}(0,1)$, and the observed binary outcome is $y_i = \mathbb{1}(y_i^* > 0)$. The source material is a 4-page PDF (reproduced in Appendix B) specifying three estimation methods for this model: (a) *maximum likelihood via Newton-Raphson*, which directly maximizes the log-likelihood using its gradient and Hessian, with the inverse Mills ratio (ϕ/Φ) for numerical stability; (b) a *Bayesian Gibbs sampler* based on the Albert-Chib data augmentation scheme (Albert and Chib, 1993), which augments the model with latent y_i^* drawn from a truncated normal and then samples $\boldsymbol{\beta}$ from the resulting conjugate normal posterior; and (c) *random-walk Metropolis-Hastings*, which proposes $\boldsymbol{\beta}$ from a random walk and accepts or rejects via the likelihood ratio.

All three methods target the same parameter $\boldsymbol{\beta}$, so their estimates should agree asymp-

totically, making correct implementation verifiable through Monte Carlo comparison. Probit estimation is also a solved problem in R: `glm(family = binomial(link = "probit"))` provides a trusted reference implementation. StatsClaw is configured to implement algorithms from scratch rather than wrap existing packages, so the `glm()` output serves as an independent benchmark against which the builder's implementation can be validated.

3.2. Setup and Prompt

The user opens Claude Code with the StatsClaw configuration loaded and provides a single prompt:

$> User Prompt

Build the R works from this PDF. Three probit estimation methods in C++ via Rcpp/Armadillo: MLE (Newton-Raphson), Bayesian Gibbs sampler (Albert-Chib data augmentation), and random-walk Metropolis-Hastings. After building, run a Monte Carlo simulation comparing all three on bias, RMSE, coverage, and computational speed across $N = \{200, 500, 1000, 5000\}$ *with 500 replications per scenario. Ship it.*

Leader resolves the target repository, detects no existing language profile, and selects the full workflow, which activates all agents including **Builder**, **Simulator**, and **Tester**.

3.3. Comprehension

Planner ingests the PDF and executes the comprehension protocol. It inventories all equations, restates each in its own notation, and identifies implementation-relevant details that the PDF leaves implicit: the inverse Mills ratio for numerical stability in truncated normal sampling, a Cholesky precomputation for the Gibbs posterior draw, and an MLE warm-start strategy for the MH chain.

The planner produces four artifacts. No clarification was needed (Hold was not raised).

`comprehension.md (excerpt)`

Latent variable model: $y_i^* = \mathbf{x}_i'\boldsymbol{\beta} + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0,1), \quad y_i = \mathbb{1}(y_i^* > 0)$.

MLE: Newton-Raphson on $\ell(\boldsymbol{\beta}) = \sum_i [y_i \log \Phi(\mathbf{x}_i'\boldsymbol{\beta}) + (1-y_i)\log(1-\Phi(\mathbf{x}_i'\boldsymbol{\beta}))]$.

Gradient uses ϕ/Φ (inverse Mills ratio) for numerical stability.

Gibbs (Albert-Chib): Augment with $y_i^* \mid y_i, \boldsymbol{\beta} \sim \mathcal{TN}(\mathbf{x}_i'\boldsymbol{\beta}, 1)$, truncated to $(0,\infty)$ if $y_i = 1$, $(-\infty, 0]$ if $y_i = 0$. Then $\boldsymbol{\beta} \mid \mathbf{y}^* \sim \mathcal{N}(\cdot,\cdot)$ via Cholesky.

Self-test verdict: FULLY UNDERSTOOD. All formulas reproduced from memory. Implicit assumption identified: MH proposal scale requires tuning for acceptance ≈ 0.40.

`spec.md` → **Builder**

Implement three C++ functions via Rcpp/Armadillo:

`probit_mle()`: Newton-Raphson, observed info, warm-start from **0**

`probit_gibbs()`: Albert-Chib, 3000 draws, 500 burn-in, Cholesky

`probit_mh()`: RW-MH, 10000 draws, adaptive proposal

R wrappers: `probit(method=...)` dispatching to C++.

`test-spec.md` → **Tester**

Validate without seeing implementation:

MLE vs `glm(link="probit")`: tol $< 10^{-6}$

Gibbs posterior mean $\to$ MLE as $n \to \infty$

MH acceptance rate $\in [0.25, 0.50]$

Edge cases: perfect separation, $p > n/10$, single predictor

Sim validation: bias $\to 0$, coverage $\to 0.95$.

3.4. Parallel Execution

Leader dispatches three agents in parallel, each receiving only its own specification. **Builder** implements three C++ estimation functions using Armadillo linear algebra, with R wrapper functions providing a consistent API. It does not know the ground-truth parameters or the DGP that will be used to evaluate its code. **Simulator** implements the DGP with known ground-truth $\beta_0 = -1$, $\beta_1 = 0.5$, runs 6,000 total fits ($4 \times 500 \times 3$ methods), and produces comparison tables. It does not know how the estimators are implemented. **Tester** validates

whether the implementation recovers ground truth from simulated data: MLE output against R's `glm()` at 10^{-6} tolerance, Bayesian posterior convergence, edge cases. It sees neither the source code nor the DGP design.

After all three pipelines complete, **Scriber** documents the process and **Reviewer** cross-audits all outputs. Upon passing tests, **Shipper** commits and pushes to GitHub upon user authorization.

✓ Pass

Pipeline isolation verified. Convergence confirmed: MLE matches `glm()` at 10^{-8}. Monte Carlo acceptance criteria satisfied (7/7). Tolerance integrity: no inflation detected. **Verdict: Pass.**

3.5. Results

Figure 2 presents the Monte Carlo results. All three methods exhibit the theoretically expected behavior: bias converges to zero (consistency), RMSE scales as $1/\sqrt{N}$ ($\sqrt{N}$-convergence), and 95% CI coverage converges to the nominal level. MLE runs in microseconds; Gibbs and MH require orders of magnitude more time but provide full posterior distributions. MH acceptance rates of 0.396–0.401 are near the $\sim$0.40 theoretical optimum.

The key observation is what the user provided versus what the system produced. The user supplied approximately 50 words of domain guidance and a 4-page PDF. The system produced a complete, installable R package with Rcpp/Armadillo C++ backends, a comprehensive test suite, a Monte Carlo simulation harness, and full documentation. Every substantive decision—which estimation methods, which C++ library, what simulation design—came from the user's prompt and the PDF. The engineering labor—code structure, build system, test infrastructure, simulation harness—was handled by the agents.

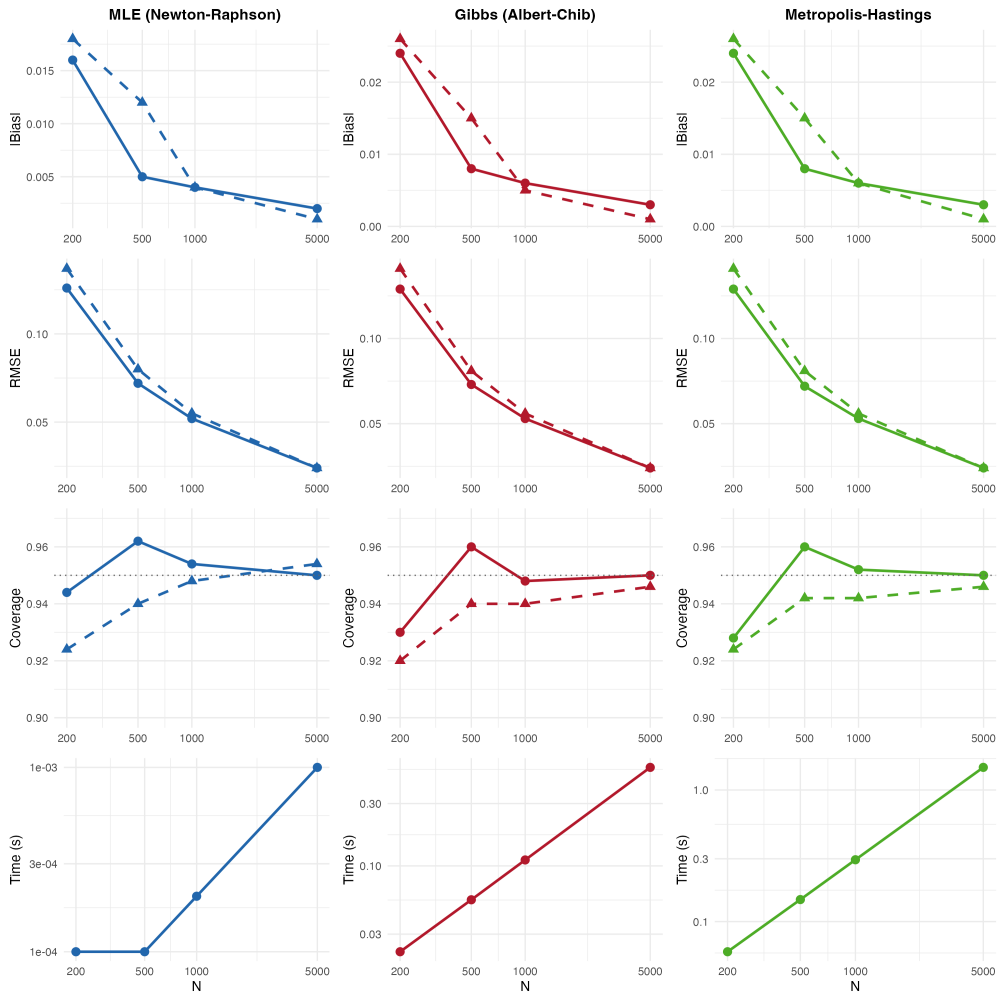

Figure 2. Monte Carlo comparison of three probit estimators across different sample sizes $N \in \{200, 500, 1000, 5000\}$ with 500 replications per scenario. Columns: MLE (blue), Gibbs (red), MH (green). Rows: |Bias|, RMSE, 95% CI coverage, computation time. All three methods exhibit consistency, $\sqrt{N}$-convergence, and nominal coverage—confirming that the C++ implementations match their mathematical specifications.

4. Practical Applications

The original motivation for developing StatsClaw was practical: the authors maintain several R packages for causal inference and panel data analysis (`interflex`, `fect`, `panelView`), and the engineering cost of keeping these packages tested, documented, and up to date had become the binding constraint on the research. We used StatsClaw to do exactly this work. The three applications below are not constructed demonstrations; they are the actual development tasks that the workflow was built to handle. Each represents a distinct

mode of statistical software development—greenfield construction, sustained refactoring, and paper-to-feature—and together they serve as direct testimony to the workflow's efficiency and effectiveness (Table 2).

TABLE 2. SUMMARY OF APPLICATIONS.

Metric	**`panelView`** (Paper→Feature)	**`interflex` Python** (Code Translation)	**`fect`** (Refactoring)
Duration	Multi-session	3 rounds	5 days, ∼20 runs
User input	Prompt + paper	∼150 words	∼10 interactions
Test suite	0 → full suite	0 → 34	131 → 590
Code scope	Refactored + new	∼3,500 lines	100+ files
Bugs found	3 (deprecation)	6 (2 silent)	11+
Key challenge	Paper comprehension	Spec. extraction	Dependencies

4.1. Paper to Feature: `panelView` Network Visualization

The `panelView` R package (Mou et al., 2024) visualizes panel data through a single-export function with approximately fifty parameters. The task required reading Correia (2016), understanding the bipartite graph representation of panel observation patterns, and adding a `type = "network"` pathway: constructing the bipartite graph, identifying singletons (degree-1 nodes), drawing convex hulls around connected components, and extending to k-partite graphs for multi-way fixed effects.

This application illustrates a distinct mode: **Planner** read a methodological paper not to implement the estimator itself, but to extract a visualization concept from the paper's exposition. The bipartite graph in Correia (2016) is explanatory apparatus, not a software specification. Translating it into a feature required comprehending the mathematics (incidence matrices, graph connectivity, iterative pruning) and then designing a user-facing API.

The planner also identified a prerequisite: the existing codebase was a monolithic 37 KB file. Adding a new dispatch pathway into this monolith would be fragile and difficult to test. The system made a proactive judgment: *refactor first, then add the feature.* The monolith was split into focused modules, a test suite was added, three `ggplot2` deprecation bugs were

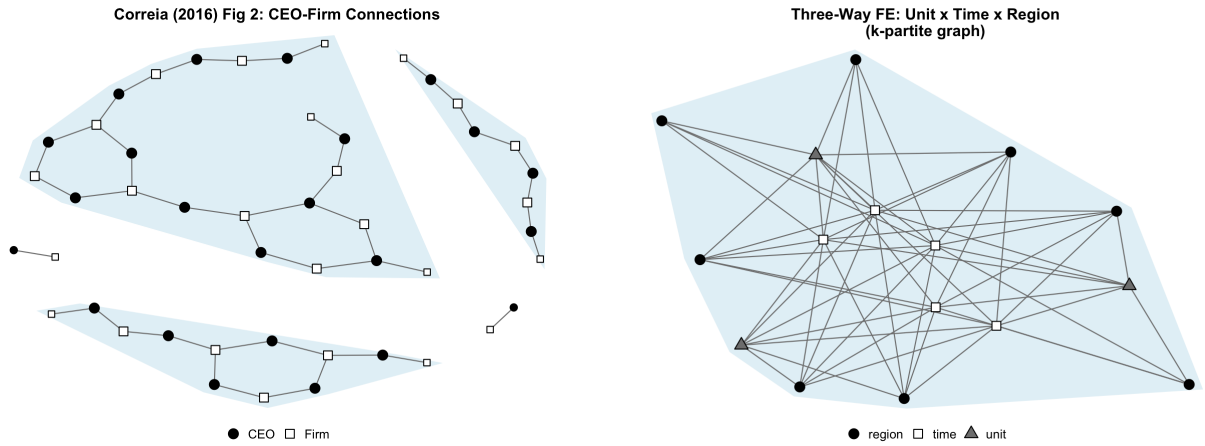

Figure 3. Left: CEO–Firm bipartite network with 48 nodes, 5 connected components, and 11 singletons, reproducing the diagnostic in Correia (2016). Right: three-way FE (unit × time × region) as a k-partite graph. Both produced by `panelview(type = "network")`.

fixed, and a 5-chapter Quarto manual replaced the stale vignette—all before the network feature was specified.

Figure 3 shows the results: a CEO–Firm bipartite network reproducing Correia (2016)'s key diagnostic, and a k-partite extension for three-way fixed effects.

4.2. Code Translation: `interflex` for Python

We constructed a complete Python `interflex` package from the R implementation—zero pre-existing Python code to a validated product with 14 modules, approximately 3,500 lines, 34 tests, and a 10-chapter Quarto tutorial.

No formal mathematical document was provided. **Planner** reverse-engineered all contracts from R source code: treatment type polymorphism (auto-detecting discrete vs. continuous treatment from the number of unique values), the DML estimation pipeline via DoubleML's influence function and B-spline best linear projection (Bach et al., 2024), and all covariance estimator variants (HC1, homoscedastic, cluster-robust CR1, PCSE).

The user provided approximately 150 words across three rounds: an initial directive to translate the R package, an API refinement (rejecting `import interflex as ifx` in favor of `import interflex; interflex(data, ...)`), and a quality assurance request.

The six-bug audit. The most significant outcome was not the initial construction but what happened after the user's third message triggered a comprehensive audit. **Reviewer**'s cross-pipeline analysis uncovered six bugs in code that already passed all 34 tests: (1) HC1 sandwich operator precedence, where NumPy's `*` binds before `@`, producing a semantically incorrect covariance computation; (2) discrete bootstrap silent fallthrough, where `elif vartype == "bootstrap": pass` silently returned delta method standard errors instead of bootstrap SEs—**silently wrong statistical results**; (3) binning weight slicing, where DataFrame operations on a NumPy array caused a crash on any weighted binning regression; (4) a dead import that caused `ImportError` on any kernel call; (5) a plotting distribution overlay that used 50 evaluation grid points instead of raw moderator values; and (6) dead prediction code that claimed B-spline re-projection but performed generic interpolation.

Figure 4 confirms visual and numerical equivalence between the R and Python implementations.

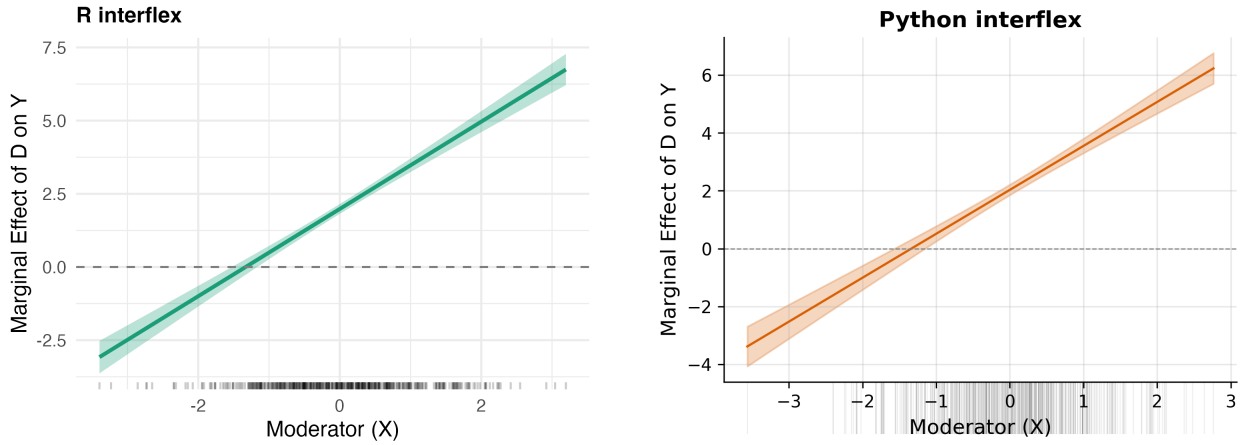

Figure 4. Marginal effect estimates from R `interflex` (left) and Python `interflex` (right) on the same DGP. Both recover the true marginal effect $\partial E[Y|D,X]/\partial D = 2 + 1.5X$ with matching point estimates and confidence intervals.

Bugs (2) and (3) would have produced silently incorrect inferential results under specific parameter combinations. Their detection by the reviewer's cross-pipeline audit—not by the test suite—is the strongest evidence in this paper for the value of separated verification. A generate-then-test workflow, where the same model writes both the code and the tests,

would likely have embedded the same misunderstanding in both.

4.3. Sustained Development: `fect`

The `fect` R package (Liu et al., 2024) implements a group of counterfactual estimators, including interactive fixed effects, matrix completion, and complex fixed effects, for causal panel analysis. A five-day refactoring campaign (approximately twenty workflow runs) addressed six interdependent workstreams: structural refactoring, cross-validation unification, convergence conditioning in C++ solvers, visualization overhaul, a 12-chapter user manual, and bug resolution. The user provided approximately ten substantive interactions covering methodology, DGP design, and documentation structure.

This is the most demanding application because changes compound: modifying a cross-validation scoring function affects parameter interfaces, bootstrap routines, plotting code, and documentation. Missing a single dependency produces bugs that may be silent for months.

Convergence conditioning. The IFE model decomposes the outcome matrix as $Y = \mu\mathbf{1}\mathbf{1}' + \alpha\mathbf{1}' + \mathbf{1}\xi' + F\Lambda' + \varepsilon$. The EM solver tracked convergence via $\|Y^{(t)} - Y^{(t-1)}\|_F / \|Y^{(t-1)}\|_F$. When μ is large (e.g., GDP data, $\mu \sim 10^4$), this global criterion is dominated by the grand mean: at `tol` $= 10^{-3}$, the solver converged with 9.4% relative error in $F\Lambda'$. **Planner** specified a two-part fix: R-level centering (subtracting $\hat{\mu}$ before the C++ call) and C++ component-wise convergence monitoring. The results were dramatic:

Component	Old error	New error	Improvement
$\hat{\alpha}$ (unit FE)	2.03×10^{-1}	3.72×10^{-3}	55×
FL' (factors)	2.99	6.89×10^{-2}	43×
$\hat{\sigma}^2$	5.07×10^{-3}	2.25×10^{-6}	2,249×

This is not just an engineering fix. Premature convergence in the EM solver propagates directly into ATT estimates, potentially invalidating published analyses that use `fect` on

data with large level effects. Figure 5 quantifies the improvement.

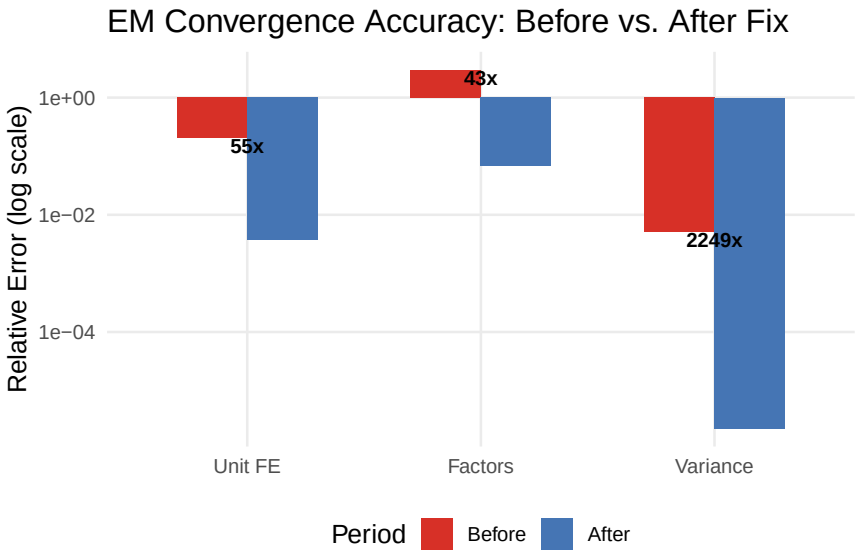

Figure 5. Component-wise relative error before and after the convergence fix (`tol` $= 10^{-3}$). The old global criterion allowed 9.4% error in the factor component because the grand mean dominated the denominator. The new criterion ensures each component converges to its own scale.

Adversarial testing as discovery. The most scientifically significant finding emerged on Day 4 of the campaign. The builder implemented a data-generating process for the complex fixed effects chapter on a balanced panel; the tester's placebo test showed no improvement from additional fixed effects. The second-dispatch builder identified the root cause: on a balanced panel, the two-way demeaning operator $M_D = I - D(D'D)^{-1}D'$ absorbs group structure entirely. If group membership is collinear with the unit dimension, the residual group mean is algebraically zero for all groups, making the extra FE a no-op regardless of confounding strength.

This mathematical insight—that balanced-panel geometry renders certain fixed-effect augmentations algebraically degenerate—was not anticipated by the planner or the builder. It was surfaced by the tester's behavioral observation:

✗ Block

Placebo test failure: adding group FE to balanced panel produces zero improvement in RMSE. Expected >5% reduction under DGP with group confounding. Suspecting algebraic degeneracy—requesting builder investigation.

The separated verification architecture did not merely catch a bug; it discovered a property of the estimator that the developers had not considered. This is the strongest qualitative evidence that adversarial verification can serve as a discovery mechanism, not just a quality assurance tool.

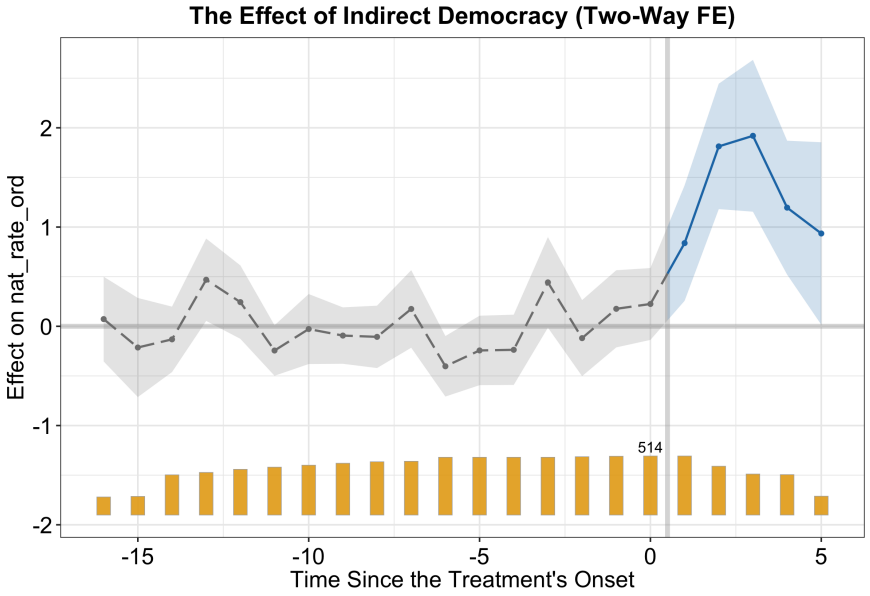

Figure 6. The effect of indirect democracy on naturalization rates (Hainmueller and Hangartner, 2019), estimated by two-way FE via `fect` (500 bootstrap replications). Grey ribbon: pre-treatment ATT centered at zero across 15 pre-treatment periods (parallel-trends validation). Blue ribbon: post-treatment effect rising to +1.8 at $t + 2$. Golden bars: number of treated units at each relative period.

Feature improvement. Figure 6 illustrates one outcome of the refactoring: the visualization of treatment effects from the Hainmueller and Hangartner (2019) study of Swiss municipalities' switch from direct to indirect democracy. Beyond the statistical results, the figure demonstrates the workflow's ability to handle delicate visual details through iterative builder–tester cycles with a human in the loop. The smooth color transition from grey (pre-

treatment) to blue (post-treatment) at $t = 0$—a rendering detail that would typically require hours of manual coding to get right—was refined through successive BLOCK and fix rounds until the visualization met the user's specifications.

Quantitative summary. Over five days: 33+ commits, tests grew from 131 to 590 (100% pass rate), 11+ BLOCK signals raised and resolved, 12-chapter Quarto manual produced, and zero known bugs at completion.

5. Discussion

The demonstration and applications tested the workflow under progressively harder conditions: from a textbook estimation problem with known ground truth (Section 3), through three practical applications (Section 4). We now assess what these experiences reveal about the workflow's strengths, limitations, and broader implications.

5.1. What Worked

Three patterns emerged consistently across applications. First, separated verification catches what testing misses. The six-bug audit in the Python `interflex` package is the clearest example: two bugs that would have produced silently incorrect statistical results survived a 34-test suite but were caught by the reviewer's cross-pipeline analysis (Figure 4). The key insight is that tests written by the same process that wrote the code share the same blind spots. Independent specification breaks this correlation.

Second, adversarial testing can serve as a discovery mechanism. The balanced-panel degeneracy in `fect` was not a bug but a mathematical property of the estimator that the developers had not considered. It was surfaced by the tester's behavioral observation through a BLOCK signal—not by deliberate investigation (Figure 5). When two independently derived specifications disagree, the disagreement reveals not only bugs but also gaps in understanding.

Third, the comprehension protocol routes human attention efficiently. The `panelView` network visualization (Figure 3) illustrates this most clearly: the planner read a methodological paper, extracted a visualization concept, and proactively refactored a monolithic codebase—all from a single user prompt. The HOLD mechanism channels user engagement toward domain decisions—which covariance estimator? which DGP design? which tolerance is scientifically meaningful?—while delegating engineering mechanics to the agents. The user's cognitive load scales with the domain complexity of the task, not with the engineering complexity. More broadly, our experience suggests that the quality of the workflow's output scales with the depth of human involvement: the more discussion between the user and the agents during planning, the better the specifications; the more the user participates in designing test criteria, the more effective the execution. The workflow substantially reduces the cost of package maintenance and new feature development, but it should not be mistaken for a tool that automatically produces perfect software from a single prompt.

5.2. Limitations

The quality of output is ultimately bounded by the underlying language model's capability. When source material is dense and highly abstract—for example, a manuscript that assumes familiarity with advanced functional analysis or measure-theoretic probability—the planner's comprehension may be shallow, producing specifications that miss subtle requirements. In such cases, performance degrades and the user must compensate with more detailed guidance. Even for material within the model's reach, methods requiring extremely deep domain-specific reasoning may exceed current capabilities, triggering repeated HOLD cycles. These cycles are a feature—they channel expert input to where it matters—but they impose a practical limit on autonomy.

More fundamentally, the adversarial architecture provides high confidence but not formal correctness guarantees. It detects bugs empirically through independent behavioral testing, not through mathematical proof. Systematically silent misunderstandings—where

the model is confidently wrong across all pipelines—remain a residual risk. The comprehension protocol mitigates this by creating a single auditable checkpoint, but it requires a domain expert who can assess whether the system's mathematical understanding is correct. Users who lack this expertise will not benefit from the protocol. The system amplifies expertise; it does not replace it.

Finally, multi-pipeline execution with independent builder, tester, and simulator agents incurs higher computational cost than single-pass code generation. This tradeoff favors correctness over speed—appropriate for statistical software where a silent bug can invalidate published analyses, but worth acknowledging as a constraint on routine use.

5.3. Implications

For methodological researchers, the engineering bottleneck documented by Ram et al. (2019) need no longer be binding. A small team—even a single researcher—can produce tested, documented, distributable software at a pace that previously required dedicated engineering resources. This does not change the science; it changes the logistics of how science reaches practitioners.

For the broader statistical computing community, the workflow's process recording creates audit trails by default (Gentleman and Temple Lang, 2007; Stodden et al., 2016). Every decision, every test result, every comprehension check is documented in a machine-readable workspace repository. This is not a substitute for formal reproducibility standards (Peng, 2011), but it provides a layer of transparency that is absent from most software development workflows.

For software citation (Smith et al., 2016), the workflow lowers the barrier to producing citable, version-controlled packages—making it easier for methodological contributions to receive credit through their software implementations.

6. Getting Started

StatsClaw requires two components: Claude Code (Anthropic's command-line agent) and a GitHub-hosted target repository with push access. No separate installation is needed—the framework is loaded as Claude Code's system configuration.

Setup. Three steps:

```
$ git clone https://github.com/statsclaw/statsclaw.git $ cd statsclaw $ claude opens
Claude Code with StatsClaw loaded
```

The agent reads the configuration automatically. Optionally, create a workspace repository (a separate GitHub repo) to store process records and handoff notes across sessions.

Usage. Provide a natural-language directive specifying the target project and desired outcome. **Leader** resolves the repository, detects the language (R, Python, Stata, Go, Rust, C/C++, TypeScript), selects the appropriate workflow, and proceeds autonomously—raising Hold signals when user input is needed.

Example prompts.

Prompt	Workflow
"Fix the failing tests in fect"	Code change
"Build a Python package from this R package"	Code + ship
"Update the README and vignette"	Docs only
"Run a Monte Carlo comparing these estimators"	Simulation study
"Read this paper and add the feature to panelView"	Code + ship

Effective patterns. Provide domain guidance, not engineering instructions. "Use HC1 with cluster-robust standard errors" is more useful than "Create a function called `compute_se()`." Review the `comprehension.md` artifact before approving code generation—a five-minute re-

view prevents hours of rework. Respond to HOLD signals with precision: "Use `lightgbm`'s defaults" is better than "Do whatever seems reasonable."

Full architecture details, the ten workflow types, and language-specific profiles are described in Appendix A.

Contributing. STATSCLAW is an open-source project, and we welcome contributions from the community. After using the workflow on your own packages, you can run the built-in `/contribute` command, which summarizes the lessons learned during your session—what worked, what required manual intervention, and what domain-specific patterns emerged—into a structured report. With your permission, these reports (`.md` files) can be submitted to the GitHub repository, where they may be absorbed into the framework's configuration to improve future performance for similar tasks. We invite researchers to visit https://statsclaw.ai/ or join us at https://github.com/statsclaw/statsclaw and contribute in whatever way they can.

References

James H. Albert and Siddhartha Chib. Bayesian analysis of binary and polychotomous response data. *Journal of the American Statistical Association*, 88(422):669–679, 1993. doi: 10.1080/01621459.1993.10476321.

Philipp Bach, Victor Chernozhukov, Malte S. Kurz, and Martin Spindler. DoubleML—an object-oriented implementation of double machine learning in Python. *Journal of Machine Learning Research*, 25(96):1–8, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Sergio Correia. A feasible estimator for linear models with multi-way fixed effects. 2016. Working paper, Duke University.

Robert Gentleman and Duncan Temple Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1):1–23, 2007. doi: 10.1198/106186007X178663.

Jens Hainmueller and Dominik Hangartner. Does direct democracy hurt immigrant minorities? Evidence from naturalization decisions in Switzerland. *American Journal of Political Science*, 63(3):530–551, 2019. doi: 10.1111/ajps.12433.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Licheng Liu, Ye Wang, and Yiqing Xu. Practical causal inference with panel data. *Journal of the American Statistical Association*, 2024. doi: 10.1080/01621459.2024.2395588. Forthcoming.

B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999. doi: 10.1257/jel.37.2.633.

Hongyu Mou, Licheng Liu, and Yiqing Xu. panelView: Visualizing panel data. *Journal of Statistical Software*, 107(7):1–20, 2024. doi: 10.18637/jss.v107.i07.

Roger D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011. doi: 10.1126/science.1213847.

Karthik Ram, Carl Boettiger, Scott Chamberlain, Noam Ross, Maelle Goldberg, and Ignasi Bartomeus. A community of practice around peer review for long-term research software sustainability. *Computing in Science & Engineering*, 21(1):59–65, 2019. doi: 10.1109/MCSE.2018.2882753.

Arfon M. Smith, Daniel S. Katz, and Kyle E. Niemeyer. Software citation principles. *PeerJ Computer Science*, 2:e86, 2016. doi: 10.7717/peerj-cs.86.

Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P. A. Ioannidis, and Michela Taufer. Enhancing reproducibility for computational methods. *Science*, 354(6317):1240–1241, 2016. doi: 10.1126/science.aah6168.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

A. StatsClaw Architecture Reference

This appendix provides the complete technical specification of the StatsClaw framework. Sections 2 and 3 of the main text describe the principles and demonstrate the workflow; this appendix documents the implementation in full.

A.1. Agent Specifications

StatsClaw comprises agents dispatched within a single Claude Code session via the built-in `Agent` tool. Table A1 summarizes each agent's role, pipeline assignment, and isolation constraints.

Table A1. Agent specifications.

Agent	Role	Pipeline	Isolation
Leader	Orchestrator	Control	N/A (dispatches only)
Planner	Requirements analyst	Bridge	Sees all sources
Builder	Code implementer	Code	Information + worktree
Tester	Independent validator	Test	Information + worktree
Simulator	Monte Carlo evaluator	Simulation	Information + worktree
Scriber	Documentation	Recording	Worktree
Reviewer	Quality gate	Convergence	Sees all pipelines
Shipper	Distribution	Ship	Sees all artifacts

Leader is the main orchestrating agent. It resolves the target repository, detects the language profile, selects the workflow type, creates the run directory, and dispatches all other agents in sequence. The leader *never* performs specialist work directly—it does not edit source files, run tests, write documentation, or interact with git. If the leader detects that a task falls within another agent's responsibility, it dispatches that agent rather than acting itself.

Planner is the sole bridge agent: the only agent with visibility into all source material. It ingests all source material—mathematical derivations, existing codebases, pseudocode, algorithm descriptions—and executes the deep comprehension protocol (Section 2.3); and produces independent specification artifacts for each downstream pipeline. The planner's output is the foundation of the entire workflow—errors here propagate everywhere. This is why the comprehension protocol is a hard gate: the planner must demonstrate understanding before producing specifications.

Builder receives only `spec.md` and implements the methodology as source code with internal unit tests. It operates in an isolated git worktree to prevent filesystem conflicts with

other agents. The builder never sees `test-spec.md` or `sim-spec.md`. Its deliverable is `implementation.md`, documenting files changed, design choices, and any unit tests written.

Tester receives only `test-spec.md` and constructs an independent validation suite from behavioral contracts, numerical tolerances, edge cases, and property-based invariants. It never sees `spec.md` or the builder's source code structure. Its deliverable is `audit.md`, containing a per-test result table, before/after comparison tables, and full command output. The tester is the only agent that can raise a BLOCK signal (Section A.3).

Simulator is activated for the full workflow. The simulator receives only `sim-spec.md` and implements data-generating processes and finite-sample performance evaluation. It treats the estimator as a black-box interface—calling the implementation without knowledge of its internal structure. Its deliverable is `simulation.md`, containing DGP implementation details, harness design, smoke-run results, and acceptance criteria assessment.

Scriber produces three mandatory artifacts after the execution pipelines complete. First, `Architecture.md` contains Mermaid diagrams (module structure, function call graph, data flow) and reference tables, written to both the target repository root and the run directory. Second, `log-entry.md` contains the complete process record: what changed, implementation notes, validation results, problems encountered, review summary, design decisions, and handoff notes. Third `docs.md` summarizes documentation changes. The scriber operates in an isolated worktree.

Reviewer The reviewer is the second agent with full cross-pipeline visibility. It reads all specification documents, all pipeline outputs, and the scriber's documentation. It performs seven checks: (1) comprehension verification, (2) pipeline isolation (verifying no agent accessed another's specification), (3) cross-comparison of specifications, (4) convergence of pipeline outputs, (5) test coverage of changed code paths, (6) tolerance integrity (ensuring the tester used exact tolerances from `test-spec.md`, with no inflation), and (7) validation evidence completeness. The reviewer issues one of three verdicts: PASS (ship safe), PASS WITH NOTE (ship with documented concern), or STOP (block and route to responsible agent).

Shipper handles all git and GitHub operations. It verifies the reviewer's PASS verdict (hard gate), confirms the target repository remote, stages code changes and user-facing documentation, commits, and pushes. It then synchronizes the workspace repository: copying `log-entry.md` to the runs archive, updating `CHANGELOG.md` and `HANDOFF.md`, and pushing. The shipper acts only upon explicit user authorization.

A.2. State Machine

Workflow progression is governed by a state machine with mandatory preconditions enforced as hard gates at each transition (Figure A1). Each transition requires specific artifacts to exist. SPEC_READY requires `comprehension.md`, `spec.md`, and `test-spec.md` from the planner. PIPELINES_COMPLETE requires `implementation.md` from the builder and `audit.md` from the tester (and `simulation.md` from the simulator, if active). DOCUMENTED requires `Architecture.md` and `log-entry.md` from the scriber. REVIEW_PASSED requires a verdict in `review.md` from the reviewer.

No state can be skipped. The leader checks preconditions before advancing and will not dispatch downstream agents until upstream artifacts are verified.

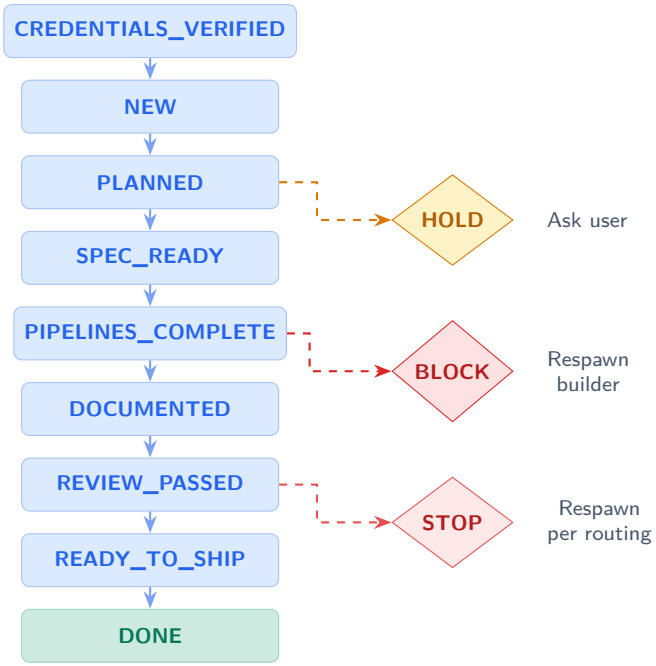

Figure A1. State machine with interrupt signals. Three signals can interrupt progression: HOLD (request user input), BLOCK (validation failure, respawn builder), and STOP (quality gate failure, respawn per reviewer routing). Each signal permits up to three retry cycles before escalating to the user.

A.3. Interrupt Signals

Three interrupt signals govern human–AI interaction and inter-agent error handling:

Each signal permits a maximum of three retry cycles before escalating to the user, preventing infinite loops while ensuring genuine recovery attempts. BLOCK signals preserve pipeline integrity: the leader re-dispatches the builder with the tester's failure details but never applies fixes directly.

Table A2. Signal handling.

Signal	Owner	Meaning	Resolution
HOLD	Planner, Builder, Scriber, Simulator	Need user input	Leader forwards question; re-dispatches on answer
BLOCK	Tester only	Validation failed	Leader re-dispatches builder with failure details
STOP	Reviewer only	Quality gate failed	Leader re-dispatches per reviewer's routing

A.4. Workflow Catalog

StatsClaw supports ten workflow types, selected by the leader based on the user's natural-language directive. Table A3 lists each type with its agent sequence.

Table A3. Workflow types and agent sequences.

#	Name	Agent Sequence
1	Full Workflow	leader → planner → [builder ‖ tester ‖ simulator] → scriber → reviewer
	Simplified Workflows	
2	Simple Code Fix + Shipping	leader → planner → [builder ‖ tester] → scriber → reviewer → shipper
3	Documentation	leader → planner → scriber → reviewer
4	Issue Patrol	leader scans issues → per issue: full code workflow
5	Single Issue Fix	leader → planner → [builder ‖ tester] → scriber → reviewer → shipper
6	Validation	leader → tester
7	Code Review	leader → reviewer
8	Scheduled Loop	leader → recurring inner workflow
9	Extremely Simple Fix	leader → builder → tester
10	Monte Carlo Exercises	leader → planner → [simulator ‖ tester] → scriber → reviewer

Brackets with ‖ indicate parallel dispatch. Workflow selection is semantic: the leader parses the user's directive and matches intent to workflow type. Manual override is available.

A.5. Runtime Artifacts

Every workflow run produces artifacts stored in a run directory within the workspace repository. Table A4 lists each artifact, its producer, and its purpose.

Table A4. Runtime artifacts produced per workflow run.

Artifact	Producer	Purpose
`request.md`	Leader	Scope, acceptance criteria, target repo identity
`impact.md`	Leader	Affected files, risk areas, required agents
`status.md`	Leader	State machine tracker
`credentials.md`	Leader	GitHub access verification
`comprehension.md`	Planner	Auditable proof of understanding
`spec.md`	Planner	Implementation specification
`test-spec.md`	Planner	Test specification
`sim-spec.md`	Planner	Simulation specification (workflows 11–12)
`implementation.md`	Builder	Files changed, design choices
`audit.md`	Tester	Per-test results, before/after comparisons
`simulation.md`	Simulator	DGP, harness design, results
`Architecture.md`	Scriber	Mermaid diagrams, reference tables
`log-entry.md`	Scriber	Complete process record with handoff notes
`docs.md`	Scriber	Documentation change summary
`review.md`	Reviewer	Cross-pipeline audit and verdict
`shipper.md`	Shipper	Git actions taken, workspace sync status

A.6. Workspace Repository

The workspace repository is a user-owned GitHub repository, separate from any target codebase, that stores all workflow artifacts and provides cross-session continuity. Its structure is:

```
workspace/
  <repo-name>/
    context.md              Repo metadata (runtime)
    CHANGELOG.md            Timeline index (pushed)
    HANDOFF.md              Active handoff (pushed)
    docs.md                 Latest doc changes (pushed)
    ref/                    Reference materials (pushed)
    runs/
      <request-id>/         Active run artifacts
      <date>-<slug>.md      Completed run logs
    logs/                   Diagnostic logs (local)
    tmp/                    Transient data (local)
```

The `HANDOFF.md` document is updated after every session. Each session's leader reads the previous session's handoff and resumes with full awareness of prior decisions, known issues, and technical insights. This eliminates the resumption cost that typically accompanies

interrupted development—particularly valuable in academic settings where development is interleaved with teaching, reviewing, and other obligations.

A.7. Language Profiles

StatsClaw auto-detects the target language from repository markers and injects language-specific conventions into all agent dispatch prompts. Table A5 lists the supported profiles.

Table A5. Supported languages and auto-detection markers.

Language	Detection Marker	Ecosystem	Validation
R	`DESCRIPTION`	CRAN	`R CMD check -as-cran`
Python	`pyproject.toml`	PyPI	`pytest`, `tox`
TypeScript	`package.json`	npm	`npm test`, eslint
Stata	`.ado` files	SSC	`do` file execution
Go	`go.mod`	Go modules	`go test ./...`
Rust	`Cargo.toml`	crates.io	`cargo test`, clippy
C	`Makefile` + `.c`	System	Unit test runner
C++	`CMakeLists.txt` + `.cpp`	System	Unit test runner

Each profile specifies validation commands, documentation conventions (e.g., roxygen2 for R, docstrings for Python), and language-specific builder and tester notes (e.g., numerical stability idioms, strictness levels for test failures).

B. Probit Specification Document

The following pages reproduce the 4-page PDF specification used as source material for the tutorial in Section 3.

Probit Model: Three C++ Estimators via Rcpp

MLE (Newton–Raphson) · Bayesian Gibbs · Metropolis–Hastings

Implementation Specification for StatsClaw Workflow

March 2026

Contents

1 Model and Notation

Latent variable formulation. For $i = 1, \ldots, N$:

$$y_i^* = \boldsymbol{x}_i'\boldsymbol{\beta} + \varepsilon_i, \quad \varepsilon_i \sim N(0,1), \quad y_i = \mathbf{1}(y_i^* > 0). \tag{1}$$

This gives $\Pr(y_i = 1 \mid \boldsymbol{x}_i) = \Phi(\boldsymbol{x}_i'\boldsymbol{\beta})$, where Φ is the standard normal CDF.

Notation. N = sample size, k = number of covariates (including intercept), $\boldsymbol{X} \in \mathbb{R}^{N \times k}$ = design matrix (row i is $\boldsymbol{x}_i'$), $\phi(\cdot)$ = standard normal PDF.

DGP for simulation. $\Pr(y = 1 \mid x_1) = \Phi(-1 + 0.5\, x_1)$, with $x_1 \sim N(0,1)$. True parameter: $\boldsymbol{\beta}_0 = (-1,\ 0.5)'$.

2 Method 1: MLE via Newton–Raphson

2.1 Log-Likelihood, Gradient, Hessian

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{N} \Big[y_i \log \Phi(\boldsymbol{x}_i'\boldsymbol{\beta}) + (1 - y_i) \log \big(1 - \Phi(\boldsymbol{x}_i'\boldsymbol{\beta})\big) \Big] \tag{2}$$

Define $q_i = 2y_i - 1 \in \{-1, +1\}$, $\eta_i = \boldsymbol{x}_i'\boldsymbol{\beta}$, and the inverse Mills ratio $\lambda_i = \phi(q_i\eta_i)/\Phi(q_i\eta_i)$.

$$\text{Gradient:} \quad \nabla \ell = \boldsymbol{X}'\boldsymbol{w}, \qquad w_i = q_i\,\lambda_i \tag{3}$$

$$\text{Hessian:} \quad \nabla^2 \ell = -\boldsymbol{X}'\boldsymbol{D}\boldsymbol{X}, \qquad d_i = \lambda_i(\lambda_i + q_i\eta_i) > 0 \tag{4}$$

2.2 Algorithm

Algorithm 1 Probit MLE

1: Initialize $\boldsymbol{\beta}^{(0)} = \mathbf{0}$
2: **for** $t = 1, \ldots, T_{\max}$ **do**
3: Compute $\boldsymbol{w}$, $\boldsymbol{D}$ from current $\boldsymbol{\beta}$
4: $\boldsymbol{\beta}^{(t)} \leftarrow \boldsymbol{\beta}^{(t-1)} - (\nabla^2\ell)^{-1}\nabla\ell$ $\quad \triangleright = \boldsymbol{\beta}^{(t-1)} + (\boldsymbol{X}'\boldsymbol{D}\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{w}$
5: Stop if $\|\boldsymbol{\beta}^{(t)} - \boldsymbol{\beta}^{(t-1)}\| < 10^{-8}$
6: **end for**
7: **Output:** $\hat{\boldsymbol{\beta}}$, $\mathrm{SE}_j = \sqrt{[(\boldsymbol{X}'\boldsymbol{D}\boldsymbol{X})^{-1}]_{jj}}$, $\ell(\hat{\boldsymbol{\beta}})$

2.3 C++ Interface

```
// [[Rcpp::export]]
Rcpp::List probit_mle(const arma::mat& X, const arma::vec& y,
                      int max_iter = 100, double tol = 1e-8);
// Returns: coefficients (vec), vcov (mat), se (vec), loglik (double)
```

Numerical stability: Use `R::pnorm(x, 0, 1, 1)` for $\log\Phi(x)$. Solve $\boldsymbol{H}\boldsymbol{s} = \boldsymbol{g}$ via `arma::solve`, not explicit inversion.

3 Method 2: Bayesian Gibbs Sampler (Albert–Chib)

3.1 Prior and Conditional Posteriors

Prior: $\boldsymbol{\beta} \sim N(\boldsymbol{\beta}_0, \boldsymbol{\Sigma}_0)$. Default diffuse: $\boldsymbol{\beta}_0 = \mathbf{0}$, $\boldsymbol{\Sigma}_0 = 100\boldsymbol{I}$.

Introduce latent $\boldsymbol{y}^* = (y_1^*, \ldots, y_N^*)'$ where $y_i^* \sim N(\boldsymbol{x}_i'\boldsymbol{\beta}, 1)$. The Gibbs sampler alternates:

Step 1 — Sample $\boldsymbol{\beta}$ (closed-form normal):

$$\boldsymbol{\beta} \mid \boldsymbol{X}, \boldsymbol{y}^* \;\sim\; N(\tilde{\boldsymbol{\beta}}, \tilde{\boldsymbol{\Sigma}}) \tag{5}$$

$$\tilde{\boldsymbol{\Sigma}} = \big(\boldsymbol{\Sigma}_0^{-1} + \boldsymbol{X}'\boldsymbol{X}\big)^{-1}, \qquad \tilde{\boldsymbol{\beta}} = \tilde{\boldsymbol{\Sigma}}\big(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\beta}_0 + \boldsymbol{X}'\boldsymbol{y}^*\big) \tag{6}$$

Derivation. The likelihood of $\boldsymbol{y}^*$ given $\boldsymbol{\beta}$ is $p(\boldsymbol{y}^* \mid \boldsymbol{\beta}) \propto \exp\big(-\tfrac{1}{2}(\boldsymbol{y}^* - \boldsymbol{X}\boldsymbol{\beta})'(\boldsymbol{y}^* - \boldsymbol{X}\boldsymbol{\beta})\big)$. Combining with the prior and completing the square in $\boldsymbol{\beta}$ gives the normal posterior above. The precision matrix is $\boldsymbol{\Sigma}_0^{-1} + \boldsymbol{X}'\boldsymbol{X}$ and the precision-weighted mean is $\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\beta}_0 + \boldsymbol{X}'\boldsymbol{y}^*$.

Step 2 — Sample each y_i^* (truncated normal):

$$y_i^* \mid (y_i = 1, \boldsymbol{\beta}) \sim \mathrm{TN}(\boldsymbol{x}_i'\boldsymbol{\beta},\ 1;\ 0,\ +\infty), \qquad y_i^* \mid (y_i = 0, \boldsymbol{\beta}) \sim \mathrm{TN}(\boldsymbol{x}_i'\boldsymbol{\beta},\ 1;\ -\infty,\ 0) \tag{7}$$

Derivation. Since $y_i^* \sim N(\boldsymbol{x}_i'\boldsymbol{\beta}, 1)$ and $y_i = \mathbf{1}(y_i^* > 0)$, the conditional $y_i^* \mid y_i$ is the original normal truncated to the region consistent with the observed y_i.

Truncated normal sampling (inverse CDF): draw $u \sim \mathrm{Unif}(0,1)$ and compute

$$y_i^* = \mu_i + \Phi^{-1}\big(\Phi(a - \mu_i) + u\,[\Phi(b - \mu_i) - \Phi(a - \mu_i)]\big), \tag{8}$$

where $\mu_i = \boldsymbol{x}_i'\boldsymbol{\beta}$ and (a, b) is $(0, +\infty)$ or $(-\infty, 0)$.

3.2 Algorithm

Algorithm 2 Probit Gibbs Sampler

Precompute $\tilde{\boldsymbol{\Sigma}} = (\boldsymbol{\Sigma}_0^{-1} + \boldsymbol{X}'\boldsymbol{X})^{-1}$ and its Cholesky $\boldsymbol{L}$ ▷ constant, compute once
Initialize $\boldsymbol{\beta}^{(0)} = \mathbf{0}$
for $t = 1, \ldots, T$ **do**
  for $i = 1, \ldots, N$ **do** ▷ Step 2: sample latent data
    $y_i^{*(t)} \sim \mathrm{TN}(\boldsymbol{x}_i'\boldsymbol{\beta}^{(t-1)}, 1;$ bounds from $y_i)$
  end for
  $\tilde{\boldsymbol{\beta}} \leftarrow \tilde{\boldsymbol{\Sigma}}(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\beta}_0 + \boldsymbol{X}'\boldsymbol{y}^{*(t)})$ ▷ Step 1: sample $\boldsymbol{\beta}$
  $\boldsymbol{\beta}^{(t)} \leftarrow \tilde{\boldsymbol{\beta}} + \boldsymbol{L}\boldsymbol{z}$, $\boldsymbol{z} \sim N(\mathbf{0}, \boldsymbol{I}_k)$
end for
Discard first B draws (burn-in). **Output:** posterior draws $\{\boldsymbol{\beta}^{(t)}\}_{t=B+1}^{T}$

3.3 C++ Interface

```
// [[Rcpp::export]]
Rcpp::List probit_gibbs(const arma::mat& X, const arma::vec& y,
                        int n_iter = 3500, int burn_in = 500,
                        Nullable<arma::vec> beta0, Nullable<arma::mat> Sigma0);
// Returns: beta_draws (mat), posterior_mean (vec), posterior_sd (vec)
```

Key optimization: $\tilde{\boldsymbol{\Sigma}}$ and $\boldsymbol{L}$ depend only on $\boldsymbol{X}$ and the prior — precompute once outside the loop.

4 Method 3: Metropolis–Hastings MCMC

4.1 Log-Posterior and Proposal

Log-posterior (up to constant):

$$\log p(\boldsymbol{\beta} \mid \boldsymbol{X}, \boldsymbol{y}) = \underbrace{\sum_i \left[y_i \log \Phi(\eta_i) + (1 - y_i)\log(1 - \Phi(\eta_i))\right]}_{\ell(\boldsymbol{\beta})} - \tfrac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}_0)'\boldsymbol{\Sigma}_0^{-1}(\boldsymbol{\beta} - \boldsymbol{\beta}_0) \tag{9}$$

Random walk proposal: $\boldsymbol{\beta}' = \boldsymbol{\beta}^{(t)} + \boldsymbol{\varepsilon}$, $\boldsymbol{\varepsilon} \sim N\big(\mathbf{0},\ s^2\,(\boldsymbol{X}'\boldsymbol{X})^{-1}\big)$.
Accept with probability $\alpha = \min\big(1,\ \exp[\log p(\boldsymbol{\beta}' \mid \cdot) - \log p(\boldsymbol{\beta}^{(t)} \mid \cdot)]\big)$.

4.2 Algorithm

Algorithm 3 Probit Metropolis–Hastings

1: Initialize $\boldsymbol{\beta}^{(0)}$ from MLE. Precompute proposal Cholesky $\boldsymbol{L}_C = \text{chol}(s^2(\boldsymbol{X}'\boldsymbol{X})^{-1})$.
2: **for** $t = 1, \ldots, T$ **do**
3: Propose: $\boldsymbol{\beta}' = \boldsymbol{\beta}^{(t-1)} + \boldsymbol{L}_C \boldsymbol{z}$, $\boldsymbol{z} \sim N(\boldsymbol{0}, \boldsymbol{I}_k)$
4: $\log r = \log p(\boldsymbol{\beta}' \mid \cdot) - \log p(\boldsymbol{\beta}^{(t-1)} \mid \cdot)$
5: If $\log u < \log r$ ($u \sim$ Unif): accept $\boldsymbol{\beta}^{(t)} = \boldsymbol{\beta}'$; else: $\boldsymbol{\beta}^{(t)} = \boldsymbol{\beta}^{(t-1)}$
6: **end for**
7: Discard burn-in. **Output:** posterior draws, acceptance rate (target: 25–40%)

4.3 C++ Interface

```
// [[Rcpp::export]]
Rcpp::List probit_mh(const arma::mat& X, const arma::vec& y,
                     int n_iter = 10000, int burn_in = 2000,
                     double scale = 1.0,
                     Nullable<arma::vec> beta0, Nullable<arma::mat> Sigma0,
                     Nullable<arma::vec> init);
// Returns: beta_draws (mat), posterior_mean (vec), posterior_sd (vec),
//          acceptance_rate (double)
```

Warm start: Initialize from MLE for faster convergence. Tune s so acceptance is 25–40%.

5 Shared Utilities (`utils.cpp`)

```
double safe_log_Phi(double x);    // R::pnorm(x, 0, 1, 1, 1) - log-space
double safe_Phi(double x);        // R::pnorm(x, 0, 1, 1, 0)
double rtruncnorm(double mu,      // Inverse-CDF truncated normal sampler
                  double lo, double hi);
```

6 Monte Carlo Simulation Study

6.1 Design

DGP	$\Pr(y = 1 \mid x_1) = \Phi(-1 + 0.5\, x_1)$, $x_1 \sim N(0, 1)$
True parameters	$\boldsymbol{\beta}_0 = (-1,\ 0.5)'$
Sample sizes N	$\{200,\ 500,\ 1000,\ 5000\}$
Replications	$R = 500$ per scenario
MLE	max 100 iter, tol 10^{-8}
Gibbs	3500 iter, burn-in 500
MH	10000 iter, burn-in 2000, $s = 1$
Prior (Gibbs & MH)	$\boldsymbol{\beta}_0 = \boldsymbol{0}$, $\boldsymbol{\Sigma}_0 = 100\boldsymbol{I}_2$

6.2 Metrics (per method, per β_j, across R replications)

$$\text{Bias}_j = \frac{1}{R}\sum_{r=1}^{R}(\hat{\beta}_j^{(r)} - \beta_{0,j}) \qquad \text{RMSE}_j = \sqrt{\frac{1}{R}\sum_{r=1}^{R}(\hat{\beta}_j^{(r)} - \beta_{0,j})^2} \tag{10}$$

$$\text{Coverage}_j = \frac{1}{R}\sum_{r=1}^{R}\mathbf{1}(\beta_{0,j} \in \text{CI}_{95\%}^{(r)}) \qquad \text{Time} = \frac{1}{R}\sum_{r=1}^{R} t^{(r)} \text{ (seconds)} \tag{11}$$

For MLE: CI from asymptotic normality $\hat{\beta}_j \pm 1.96 \cdot \text{SE}_j$. For Gibbs/MH: CI from 2.5% and 97.5% quantiles of posterior draws.

6.3 Acceptance Criteria

C1 MLE coefficients match `glm(..., family=binomial(link="probit"))` within 0.05

C2 Gibbs and MH posterior means within 0.1 of MLE for $N \geq 500$

C3 MH acceptance rate between 20% and 50%

C4 95% CI coverage between 90% and 99% for all methods

C5 MLE at least 5× faster than Gibbs (C++ vs C++)

C6 `R CMD check` passes with 0 errors, 0 warnings

C7 Package installs via `R CMD INSTALL` and all functions callable from R