Redesign Memo: *An Introduction to Probability with R* (Bootcamp)

Internal note

January 17, 2026

Context and constraints. Audience: heterogeneous, mid-career professionals (finance and adjacent technical backgrounds) entering a Master's program in quantitative/computational finance. Format: five online units of 3 hours, spread over roughly one month (January). The course has a dual mandate: (i) teach probability foundations in a finance-relevant way, and (ii) give students a first serious encounter with R (while other courses primarily use Python). Current course site: `https://martin-summer-1090.github.io/Probability_Introduction/`.

High-level design goal (what "success" should look like)

After five sessions, every student should be able to:

- translate a finance-flavored uncertainty question into a *computational experiment*;
- implement it in *base R* (objects, indexing, functions, simulation);
- quantify uncertainty (Monte Carlo error, sensitivity checks);
- communicate results reproducibly (a small Quarto report).

In the LLM era: syntax is cheap; **modeling, verification, and interpretation are expensive**.

What I would keep (core strengths to preserve)

- **Simulation-first intuition**: build probability concepts by constructing random experiments in code.
- **Finance-first motivation**: keep examples that feel like real finance problems (risk, dependence, fraud detection, identification/cryptography).
- **Projects between sessions**: the "do it yourself" arc is essential for heterogeneous cohorts.
- **Explicit LLM use**: keep it, but frame it as *code generation + code auditing*, not outsourcing thinking.

The main change from scratch: one coherent storyline

Rather than five topic blocks, design the course as one continuous build:

> *"We are building a minimal risk engine: define uncertainty → generate scenarios → measure risk → stress assumptions → report."*

Each session adds one indispensable capability to that engine. This reduces cognitive context switching for weaker students and prevents stronger students from feeling the course is a sequence of disconnected mini-worlds.

Second big change: two rails (Core vs. Extension)

To handle heterogeneity without splitting the class socially:

- **Core rail (required)**: base R only; minimal syntax; deliverables that everyone can achieve.
- **Extension rail (optional)**: for fast learners—performance, nicer visualizations, extra modeling depth, optional packages.

Rule of thumb: the core rail must be runnable by a tired student at 22:30 on a laptop.

Base-R-first policy (and why)

- Teach durable concepts: objects, indexing, vectorization, functions, randomness, checks.
- Use packages only when they introduce a *new capability*, not merely a new syntax.
- Offer a short "translation appendix" later (base manipulation $\leftrightarrow$ tidy pipelines), but do not make it the primary language of explanation.

What I would move out of the critical path

In a five-session bootcamp, the following are valuable but should be *optional* or deferred:

- deeper R internals (environments/scoping beyond what is needed for functions);
- performance engineering beyond "vectorize first, then measure";
- parallel computing (mention as a next step; do not require it).

One "cutting-edge but minimal" addition: LLM-generated code audits

Add a recurring exercise type: students receive a short code snippet (possibly LLM-generated) and must answer:

1. What does it compute, in plain English?
2. What assumptions does it make (distribution, independence, parameterization)?
3. What checks would you add (invariants, dimensions, edge cases)?
4. What counterexample could break it or mislead interpretation?

This directly trains the professional skill they will need most: *reading and validating code.*

A concrete 5×3h blueprint (base-R core)

1. **Random experiments & simulation primitives.** Coin toss $\rightarrow$ vectors $\rightarrow$ empirical frequencies; seeds; first plots (base). *Project: collisions / identifiers* (hook: crypto/payment systems).
2. **Rules of probability as computational rules.** Events as sets; complements/unions; LLN intuition via simulation; independence as a modeling choice. *Project: Benford-style forensics* (hook: "does this data smell funny?").
3. **Dependence is the whole game in finance.** Conditional probability; Bayes updating; joint vs marginal; illustrate "independence error" costs. *Project: discrete loss model under alternative dependence assumptions.*

4. **Random variables as models you can interrogate.** Expectation/variance/covariance; sampling distributions; binomial model as workhorse. *Project: binomial lattice implemented as clean functions + tests.*

5. **Continuous models & Monte Carlo risk reporting.** Normal/lognormal as approximations; quantiles; VaR/ES intuition; Monte Carlo engine; reporting in Quarto. *Extension: performance (vectorization, profiling), then "what changes at scale".*

Minimum tooling (modern, but not heavy)

- **One blessed workflow:** RStudio Project + Quarto + simple folder convention.
- **Reproducibility habits:** `set.seed()`, session info, and a "run-all" script per project.
- **Assessment artifacts:** each project ends with (i) a short Quarto report, (ii) one figure, (iii) one sanity-check table.

Implementation checklist (pragmatic)

- Pre-course: a 20-minute "setup + first success" onboarding (run R, run a script, render a Quarto doc).
- Each session: 20–30 min concept, 40 min live coding, 30 min breakout practice, 20 min debrief, 10 min "what to do next".
- Between sessions: small, sharply scoped projects; provide a minimal template; publish a worked solution after submission.

Bottom line. Keep the finance-driven, simulation-driven identity. Tighten the arc into a single narrative (a minimal risk engine), teach on two rails to handle heterogeneity, go base-R-first, and explicitly train verification (including LLM code audits). This shifts weight from syntax to computational thinking—exactly what the cohort needs.