Supplementary Note 2

Conversion Record: the R Package `rpart` to Python

Yufei Cai Jun Li

About this document. This is a contemporaneous engineering record of the `rpart` conversion reported in *r2py: AI-Assisted Conversion of R Statistical Packages to Python*. It documents each phase as it was carried out, including diagnostic paths through defects that were subsequently fixed, and is supplied so that the process described in the main text can be audited. It is not intended to be read end to end; Supplementary Information gives an index from claims in the main text to the evidence for them.

Phase numbering. This record numbers phases in the order the work was *executed*, 1 through 13. The main text numbers them by *structure*: an invariant seven-phase core preceded by a prologue whose depth is set by the foreign-function interface. The two schemes describe the same phases:

This record	Main text
Phases 1–5	Prologue P1–P5
Phases 6–12	Core phases 1–7
Phase 13	Coda (presentation surface)

Header counts. This record refers to *53 header files*; the main text refers to *51 R C API symbols*. Both are correct: the 51 items are implemented one header apiece, and the directory additionally contains the master include `fake_R.h` and the arena allocator `fake_arena.h`.

Contents

Abstract

This report documents the systematic conversion of the R package `rpart` (version 4.1.27)—the reference implementation of the CART (Classification and Regression Trees) methodology of Breiman, Friedman, Olshen, and Stone (1984)—into an installable Python package named `r2py_rpart`. Rather than reimplementing the algorithms from scratch, the project reuses `rpart`'s original C source code directly, wrapped behind a layer of fake R C API headers that removes the runtime dependency on `libR.so`, and converts the R source layer to Python function by function in a dependency-aware order guided by 172 machine-generated language-construct conversion guides. The work documented here spans Phases 1 through 13: structural analysis of the C and R source layers (Phases 1–2 and 6), generation and compilation verification of the fake-header infrastructure (Phases 3–5), establishment of the Python build and CI/CD pipeline (Phase 7), cataloging of base-R language dependencies and their conversion guides (Phase 8), automated R-to-Python function conversion, package assembly, and FFI bridging for the user-defined-split callback path (Phase 9), a full R-versus-Python behavioral equivalence audit and bug-fix pass culminating in a working user-defined-split callback bridge (Phase 10), conversion of `rpart`'s own historical R test suite together with a forensic investigation that traced a latent cross-validation discrepancy to a single control-parameter default bug (Phase 11), systematic, R-benchmarked test generation and bug resolution for every function in the package's public interface (Phase 12), and end-to-end translation of two example workflows accompanied by a round of rendering-parity fixes to the plotting stack (Phase 13). The resulting package builds and installs under Python 3.14 in the `r-to-python` conda environment on a Red Hat Enterprise Linux 9 development node; its test suite grew from an initial 11-test regression suite covering the primary `rpart()` entry point to 846 tests spanning the full public interface, of which 844 pass and the remaining two are permanently marked as documenting a single, structurally unfixable formatting divergence from R.

1 Introduction

The `rpart` package is one of the most widely used machine learning libraries in the R ecosystem. It implements the CART (Classification and Regression Trees) methodology introduced by Breiman, Friedman, Olshen, and Stone (1984), providing decision tree fitting for continuous, categorical, count, and survival responses through a unified interface. As a *recommended* package, `rpart` ships with every R installation and is depended upon by hundreds of CRAN packages. Despite the centrality of decision trees to modern machine learning, no faithful Python port of `rpart` exists that preserves its full feature set: surrogate splits for missing-data handling, four built-in splitting methods, a user-defined split callback mechanism, cost-complexity pruning, and the complete cross-validation and visualization infrastructure.

This project undertakes the systematic conversion of `rpart` version 4.1.27 to Python, targeting behavioral equivalence with the original R package. The target Python package is named `r2py_rpart`. Rather than reimplementing the algorithms from scratch—which would introduce independent bugs and divergences from the canonical implementation—the strategy is to reuse the original C source code directly and to convert the R source layer function by function in a structured, dependency-aware manner.

The conversion strategy has two parallel tracks. The first track concerns the compiled C back-end: `rpart`'s five C entry points (`C_rpart`, `C_pred_rpart`, `C_xpred`, `C_rpartexp2`, `C_init_rpcallback`) are wrapped with a layer of fake C++ headers that replace the R C API (normally provided by `libR.so`) with standalone implementations, enabling the original C code to be compiled as a shared library and called from Python via `cffi`. The second track concerns the R source layer: all 47 functions across 36 R source files are analyzed for their dependency structure and converted to Python one by one in a topologically sorted order, with each conversion guided by pre-generated Markdown guides for the 172 distinct base-R language constructs used in the package.

The entire conversion process is divided into thirteen sequential phases. Phases 1–2 statically analyze the existing C source layer: its internal call graph (Phase 1) and the R C API surface it references (Phase 2). Phases 3–5 build the C compilation infrastructure on top of that analysis—implementation guides and generated fake C++ headers that replace the R C API (Phases 3–4), followed by pure-C entry-point wrappers and the bulk migration of the original C sources onto the fake headers (Phase 5). Phase 6 performs the analogous structural analysis of the R source layer. Phase 7 establishes the Python build and CI/CD infrastructure. Phase 8 catalogs every base-R language construct used in the package and generates a machine-readable conversion guide for each. Phase 9 drives the R-to-Python function conversion, package assembly, and the `cffi`-based FFI bridge for the user-defined-split callback path. Phase 10 performs a full R-versus-Python behavioral equivalence audit and bug-fix pass. Phase 11 converts `rpart`'s own historical R test suite and traces a latent cross-validation discrepancy to its root cause. Phase 12 generates and resolves R-benchmarked tests covering every function in the package's public interface. Phase 13 translates two end-to-end example workflows and resolves the visual rendering-parity issues they expose. This report documents each phase in order, describing both what was accomplished and the technical reasoning

behind the design choices made.

1.1 Motivation for a Standalone Python Port

Beyond the simple absence of a native Python port, a further motivation for this project is the limitation of existing runtime-bridging solutions. Tools such as `rpy2` and R's `reticulate` package can call R functions from Python (or vice versa) at runtime, but both require a full R installation to be present alongside the Python interpreter, incur inter-process communication overhead on every call, and complicate containerized or otherwise self-contained deployment. For applications that demand portability—embedding a decision-tree fitting routine inside a pure-Python data pipeline, a compiled binary distribution, or a serverless function, for instance—a self-contained port that requires neither an R runtime nor any external bridging layer is strongly preferable to a bridge that merely forwards calls to a live R process.

Manual porting is the traditional remedy for this gap, but it does not scale to a package of `rpart`'s size, and it does not by itself eliminate the risk of introducing bugs that diverge from the canonical R behavior. Language-specific conventions that are invisible from source inspection alone can silently corrupt output without raising any exception: R's 1-based indexing versus Python's 0-based indexing, R's `missing()` lazy-argument-detection semantics versus Python's eager default-argument evaluation, R's column-major array storage versus NumPy's row-major default, and R's S3 object system versus Python's lack of an equivalent implicit dispatch mechanism are all potential sources of divergence that a translator—human or automated—must handle deliberately rather than by direct transliteration. Large language models (LLMs) offer a path toward automating this translation, but an LLM operating independently on each function, with no shared, explicit record of how such constructs were handled elsewhere in the same package, risks treating the same underlying semantic gap inconsistently across different functions—producing code that is individually plausible yet collectively incoherent, and correspondingly difficult to debug once a divergence is eventually found. The conversion strategy adopted in this project—reusing the original, validated C source verbatim rather than reimplementing the CART algorithm from scratch, and generating a machine-readable catalog of every base-R language construct and its precise Python translation before any function is converted—is designed specifically to contain both of these risks: the C-reuse strategy confines the entire correctness burden to the narrow, five-entry-point R-to-C interface described below, and the pre-generated language-dependency guides of Phase 8 ensure that a construct such as `tapply` or `format` is translated identically wherever it recurs across the package's 36 R source files, regardless of which conversion-agent invocation encounters it or in what order.

1.2 Package Architecture

`rpart` follows a three-tier architecture. The public R API (Tier 1) provides formula-based model fitting, S3 method dispatch, and diagnostic utilities. The R infrastructure layer (Tier 2) handles data preparation, method-specific initialization, coordinate geometry for tree plotting, and numeric formatting. The C back-end (Tier 3) performs the computationally intensive work: greedy recursive partitioning, surrogate-split search, cross-validation across

the full complexity-parameter sequence, and prediction routing. The C code is organized around a dispatch table (`func_table.h`) that maps method integers to four function-pointer slots (`init_split`, `choose_split`, `eval`, `error`), keeping all bookkeeping logic completely method-agnostic. Method integer 4 is reserved for user-defined (R-level callback) splits, implemented via the `rpartcallback` mechanism.

A key structural observation motivating the conversion strategy is that the R-to-C interface is narrow: only five `.Call()` invocations exist across all 36 R source files. This means that if these five entry points can be faithfully exposed through a Python-callable interface, the R source layer can be converted independently without touching the C layer.

1.3 Repository Structure

The working directory `python-rpart/` is the working tree of the `python-rpart` repository hosted at `github.com/r2py-project/python-rpart`. The `r2py_rpart/` subdirectory is additionally managed as a git subtree linked to a separate repository at `github.com/r2py-project/r2py_rpart`, allowing the Python package to be versioned and published to PyPI independently of the broader project. The cluster scripts `git_pull.sh` and `git_push.sh` at the project root encapsulate the subtree synchronisation commands required to keep both remotes up to date from the HPC node.

Path	Contents
`rpart/`	Unmodified R package source (`rpart` version 4.1.27), as obtained from CRAN; the canonical reference against which every later phase is checked.
`r2py_rpart/`	Git subtree linked to `github.com/r2py-project/r2py_rpart`; the installable Python package. Internal structure detailed in Section 1.4.
`c_refactor_analysis/`	C dependency-graph analysis artifacts from Phase 1 (`src/`, `lib/`).
`r_extern_analysis/`	R external item extraction CSVs, fake-header implementation guides, and fake-header conversion guides from Phases 2–4 (`src/`, `fake_guides/`, `conversion_guides/`).
`structural_analysis/`	R structural dependency analysis JSONs and dependency-graph artifacts from Phase 6 (`R/`, `lib/`).
`language_dependency_analysis/`	Per-file language-dependency CSVs and the 172 conversion guides from Phase 8 (`R/`, `conversion_guides/`).
`conversion_results/`	Per-function JSON translation artifacts from Phase 9.1 (`R/`).
`docs/`	Architecture notes, planning PDFs, per-phase summaries in `daily_summaries/`, and this report in `report/`.
`.claude/agents/`	Sub-agent specification Markdown files.
`.claude/commands/`	Skill (slash command) specification Markdown files.
`git_pull.sh`	Cluster batch script; pulls from the `python-rpart` main remote and merges changes from the `r2py_rpart` subtree remote into `r2py_rpart/`.
`git_push.sh`	Cluster batch script; pushes to the `python-rpart` main remote and propagates `r2py_rpart/` changes to the `r2py_rpart` subtree remote via `git subtree push`.
`install_environments.sh`	Cluster batch script; provisions the `r-to-python` conda environment (`meson-python`, `numpy`, `pandas`, `rpy2`, `r-rpart`, and related packages) used throughout every phase of this report.

1.4 Target Package Layout

The `r2py_rpart` Python package is organized as follows.

Path (relative to `r2py_rpart/`)	Contents
`src/`	Copy of the original C source files from `rpart/src/`, modified to replace real R API includes with fake headers (5 of 35 files modified; Phase 5).
`r_fake_headers/`	The 53 fake C++ header files that replace R's C API, eliminating the runtime dependency on `libR.so` (Phases 3–4).
`c_entry_points/`	Six pure-C wrapper files translating between the Python-callable (plain-array) interface and the SEXP-based internal interface: five `.Call` entry-point wrappers (Phase 5) plus the Category E interpreter-item helper file (Phase 9.3).
`r2py_rpart/`	The Python package itself: 36 `.py` modules converted from the R source (Phase 9), plus `__init__.py`, which loads `_rpart_core.so` via `cffi` and exposes the low-level C entry points as private symbols.
`meson.build`	Meson build definition: compiles `src/` and `c_entry_points/` into `_rpart_core.so` and enumerates every `.py` module for `py.install_sources()` (Phases 7, 9.2).
`pyproject.toml`	Package metadata, build-system declaration (`meson-python`), and the `[tool.cibuildwheel]` platform configuration (Phase 7).
`tests/`	The `pytest` suite, including `test_rpart.py` (Phase 9.2–9.3).
`examples/`	Two paired end-to-end R/Python example workflows and their fixture data: `regression_tree.{R,py}` (with the `income.data` fixture) and `classification_tree.{R,py}` (with the `iris_jun.{RData,csv}` fixture), together with the `*_R`/`*_Python`-suffixed PDF plots and `*_output_{R,Python}.txt` console-output pairs each script produces, used for the side-by-side rendering-parity comparison in Phase 13.
`pip_install.sh`	Cluster batch script for a non-editable development install via `pip install -no-build-isolation` (Phase 7).
`.github/workflows/python-publish.yml`	GitHub Actions workflow; builds wheels for five platforms and publishes them to PyPI on every release via OIDC trusted publishing (Phase 7).

1.5 Automated Workflow Infrastructure

This project employs **Claude Code** (Anthropic) as an AI-assisted development environment. Recurring multi-step tasks are encoded as *skills*—top-level slash commands whose specifications are stored as Markdown files under `.claude/commands/`. Skills orchestrate one or more *sub-agents*, whose specifications are stored under `.claude/agents/` and which are dispatched via the `@agent-name` notation from within a skill. Both skills and sub-agents are specified as Markdown files with YAML front matter declaring a name and description, followed by a structured natural-language body describing execution steps and output schemas.

Throughout this report, a skill invocation rendered as `/analyze-c-folder-dependencies` (see Appendix A.1) links to that skill's full specification in the appendix; the sub-agent(s) it dispatches are referenced separately by their `@agent-name` and are likewise reproduced in full there. At the end of each working session, the `/summarize-research-report` skill was invoked to synthesize that session's activities into a structured Markdown summary, saved to `docs/daily_summaries/`; those summaries are the primary source material for this report.

2 Phase 1: C Source Dependency Analysis

2.1 Motivation

Before any code can be converted or the C compilation infrastructure can be designed, it is necessary to understand the internal structure of the C source layer: which functions call which other functions, and which functions call into the R C API. This structural knowledge serves two purposes. First, it provides the call graph needed to determine the correct bottom-up order for conversion (leaf functions with no internal dependencies can be translated first, followed by their callers). Second, it identifies exactly which R API symbols are used, which is the foundation for the fake-header generation in later phases.

A single agent invocation reading all 35 C and header files simultaneously and reasoning about the complete call graph in one pass would be the simplest design, but it does not scale: the combined source exceeds what a single context window can hold together with the reasoning needed to correctly attribute every call site, and a single overloaded invocation is also harder to audit than many small, independently checkable ones. The `/analyze-c-folder-dependencies` (see Appendix A.1) skill therefore dispatches one sub-agent invocation per file—each responsible for exhaustively reading a single file (and its transitively included local headers) and classifying every call it contains—rather than one invocation over the whole directory. This per-file, sequential dispatch pattern, established here for the C source layer, recurs throughout the project wherever a package must be exhaustively cataloged before any conversion begins: Phase 2's R-external-item extraction and Phase 6's R structural analysis both follow the same pattern, for the same context-window and auditability reasons.

2.2 Methodology

The `/analyze-c-folder-dependencies` (see Appendix A.1) skill was invoked against the `rpart/src/` directory, which contains all 35 C and header files of the package:

```
/analyze-c-folder-dependencies rpart/src/
```

This skill dispatched the `@analyze-c-file-dependencies` sub-agent sequentially to each file. Each agent read the full source text of its target file and all transitively included local headers, then classified every function call as either an *internal dependency* (resolved within the project) or an *external dependency* (R API call such as `R_alloc`, `PROTECT`, `error`). The output was a JSON file per C source, saved to `c_refactor_analysis/src/` with the naming convention `<filename>.json`.

Following the automated analysis, a manual cross-examination of all 35 JSON files revealed three inconsistencies that were corrected:

1. `xpred.c.json`: the function pointers `rp_init` and `rp_eval` were missing from `xpred`'s internal dependencies despite being called via `(*rp_init)(...)` and `(*rp_eval)(...)`.
2. `rpart_callback.c.json`: the `_` macro was incorrectly listed as a direct internal dependency of `init_rpcallback`; it is only called inside `compat_getVar`, not `init_rpcallback` itself.
3. `rpart_callback.c.json`: the `_` macro entry lacked its `dgettext` external dependency, which was correct in the analogous entry in `rpart.h.json`.

Running the graph-construction script `dependency_graph.py` after corrections revealed five identifiers referenced as internal dependencies but absent as defined entries in any file: the four `EXTERN` function pointer variables `rp_init`, `rp_choose`, `rp_eval`, `rp_error` (declared in `rpart.h`) and the `static` function pointer `impurity` (in `gini.c`). These are runtime-dispatch function pointer variables, not function definitions; they were added with empty dependency lists to resolve the dangling graph nodes.

The `networkx` package (version 3.6.1) was installed into the `r-to-python` conda environment as a missing dependency before the final script runs.

2.3 Key Findings

The final dependency graph contains 136 nodes (35 file nodes, 66 function/macro nodes, 35 external-dependency nodes) and 255 edges. The topological level distribution in `dependency_levels.csv` shows 38 nodes at level 0 (entry points or leaves with no unresolved callers), 24 at level 1, and 4 at level 2. The four level-2 nodes—`ALLOC`, `rp_error`, `compat_getVar`, and `branch`—are the topmost callers in the graph. Of the 66 function nodes, 47 are pure leaves that make no internal calls. Structurally, `rpart.c` is the primary integration point (12 internal dependencies, 15 R API calls), while `rpart_callback.c` is architecturally isolated—it does not include `rpart.h` and implements its own version-compatibility logic for R $< 4.5.0$ vs. R $\geq 4.5.0$ using its own `compat_getVar` shim. `xpred.c` (`xpred`) and `xval.c` (`xval`), the

package's two cross-validation entry points, both dispatch through the `rp_init` and `rp_eval` function pointers rather than calling a method's functions directly—the same indirection that produced the two missing-dependency corrections to `xpred` described in Step 2 above.

3 Phase 2: R External Item Extraction from C Sources

3.1 Motivation

Having established the internal call graph in Phase 1, the next prerequisite for the C compilation strategy is a precise inventory of every R API symbol used by the package: which macros, types, and functions from R's header files appear in the C source, in which files, at which line numbers, and in what syntactic context. This inventory serves as the input to the fake-header generation pipeline: each unique R external item requires its own fake implementation. Without a complete and accurate inventory, fake headers would be incomplete and the compilation would fail at unpredictable locations.

3.2 Methodology

The `/extract-c-folder-r-extern-items` (see Appendix B.1) skill was invoked against `rpart/src/`:

```
/extract-c-folder-r-extern-items rpart/src/
```

The `@extract-c-file-r-extern-items` sub-agent was dispatched for each of the 35 C and header files—one invocation per file, for the same context-window and auditability reasons given in Phase 1 (Section 2), and additionally because this extraction task requires the agent to hold a single file's full source in view in order to correctly attribute every symbol reference to its precise enclosing context statement. Each agent read the source file and all transitively included local headers, then searched the R include directory at `~/.conda/envs/r-to-python/lib/R/include/` to determine whether each non-standard identifier was declared in an R header. A critical disambiguation was applied throughout: local wrapper macros defined in `rpart.h` itself (`ALLOC`, `CALLOC`, `RPARTNA`, `LEFT`, `RIGHT`) were consistently excluded, since they are declared in the project header, not in the R include directory, even though they wrap R API calls (`R_alloc`, `R_chk_calloc`, `ISNAN`).

Per-file CSVs were saved to `r_extern_analysis/src/` with the schema `external_item, header_file, category, file_name, line_number, context_statement`. A combination script `r_extern_analysis/combine_csvs.py` merged all files into a single sorted master table `r_extern_analysis/combined.csv`.

3.3 Key Findings

The combined table contains 235 reference rows across 51 unique R external identifiers drawn from 16 source files (the remaining 19 files rely exclusively on project-internal headers and standard C primitives). The 51 identifiers are drawn from 10 R include headers, with

`Rinternals.h` accounting for 165 of the 235 references. The category breakdown is 184 functions (78%), 39 types (17%), and 12 variables (5%).

R Header	Reference Count
`Rinternals.h`	165
`R_ext/Print.h`	28
`R_ext/Error.h`	9
`R_ext/Boolean.h`	7
`R_ext/Rdynload.h`	6
`R_ext/RS.h`	6
`R_ext/Arith.h`	5
`Rversion.h`	4
`R.h`	3
`R_ext/Utils.h`	2

Two files stand out architecturally. `rpart.c` and `xpred.c` are the most R-API-dense, making heavy use of `SEXP`, `PROTECT`/`UNPROTECT`, `allocVector`/`allocMatrix`, `INTEGER`, `REAL`, and list-construction functions. `print_tree.c` contributes 26 rows from a single R external item (`Rprintf`), the highest single-function repetition in the dataset, reflecting its role as the package's plain-text tree-printing routine. `rpart_callback.c` is uniquely notable: it is the only file that calls R interpreter-level functions (`eval`, `findVar`, `findVarInFrame`, `install`, `R_getVar`). These five symbols are required only when `rpart()` is called with a user-defined splitting method (`method=4`); all four standard methods (anova, poisson, class, exp) execute without them. This observation will become the basis for the Category E design in Phase 3.

4 Phase 3: Fake C++ Header Implementation Guides

4.1 Motivation

The central challenge in making the rpart C source files callable from Python without `libR.so` is that they include R's C API headers and call R's C API functions. These headers and functions are not available without a running R interpreter. The solution adopted in this project is to write *fake* C++ headers that provide compilable, self-contained implementations of all 51 R external items identified in Phase 2. When the C source files include `fake_R.h` instead of the real R headers, they can be compiled and linked as a standalone shared library.

Before writing the headers themselves, it is essential to plan the implementation strategy for each item. Memory management, type representations, and singleton semantics all require careful design to ensure that the 36 source files compile together without ODR (One Definition Rule) violations or linkage conflicts. Phase 3 produces one Markdown guide per external item, documenting the full implementation blueprint.

4.2 Methodology

The `/generate-r-extern-fake-guides` (see Appendix C.1) skill was invoked with the master CSV as input:

```
/generate-r-extern-fake-guides rpart/src/ r_extern_analysis/combined.csv
   r_extern_analysis/fake_guides/
```

The `@generate-r-extern-fake-guide` sub-agent was invoked sequentially for each of the 51 unique external items, in the order they appear in the pre-sorted CSV. Sequential processing is mandatory here because foundational definitions (the `SEXP` struct layout, the `SEXPTYPE` enum, the arena allocator) must be established before dependent items (`INTEGER`, `allocVector`, `error`) can reference them. Each agent read the relevant rpart source files and the real R include headers, and produced a Markdown guide named `<external_item>.md` in `r_extern_analysis/fake_guides/`. The 51 resulting guides total approximately 1.07 MB, averaging roughly 21.5 KB per guide.

4.3 Classification of External Items

The 51 guides were classified into five categories based on the fake implementation strategy required:

Category	Description	Items
A (19)	Type, enum constant, or registration no-op	`SEXP`, `INTSXP`, `REALSXP`, `STRSXP`, `VECSXP`, `DL_FUNC`, `DllInfo`, `R_CallMethodDef`, `Rboolean`, `TRUE`, `FALSE`, `R_VERSION`, `R_Version`, `R_NilValue`, `R_NamesSymbol`, `R_UnboundValue`, `R_forceSymbols`, `R_registerRoutines`, `R_useDynamicSymbols`
B (16)	Accessor macro or inline function	`INTEGER`, `REAL`, `CHAR`, `PROTECT`, `UNPROTECT`, `LENGTH`, `PRINTNAME`, `ISNAN`, `R_FINITE`, `asInteger`, `asReal`, `isReal`, `ncols`, `nrows`, `SET_STRING_ELT`, `SET_VECTOR_ELT`
C (7)	Allocation or memory management	`allocVector`, `allocMatrix`, `R_alloc`, `R_chk_calloc`, `R_Free`, `mkChar`, `setAttrib`
D (4)	Error, warning, or print function	`error`, `warning`, `Rprintf`, `R_CheckUserInterrupt`
E (5)	R interpreter item (Python function-pointer bridge)	`eval`, `findVar`, `findVarInFrame`, `install`, `R_getVar`

4.4 Key Technical Decisions

Memory model. The guides establish a two-tier memory model. Scratch allocations via `R_alloc`/`ALLOC` delegate to a per-frame arena allocator (`ArenaFrame`) that is freed

automatically when the `.Call` boundary wrapper returns. Persistent allocations for `SEXP` nodes and their data buffers live on the process heap and are freed explicitly via `R_Free` or `free_sexp()`. The same heap tier also covers the package's own `Node` and `Split` structs, which are allocated via `CALLOC` (a local wrapper around `R_chk_calloc`) and persist for the lifetime of the call rather than being scoped to the arena.

R version pinning. `R_VERSION` must be faked as `R_Version(4,4,0) = 263168`. This value simultaneously satisfies two conflicting preprocessor guards: `R_VERSION >= R_Version(2,16,0)` in `init.c` (enabling the `R_forceSymbols` branch) and `R_VERSION < R_Version(4,5,0)` in `rpart_callback.c` (selecting the `compat_getVar` shim instead of the native `R_getVar` symbol, which would require `libR.so`).

`R_getVar` is a macro, not a symbol. For R $< 4.5.0$, `R_getVar` expands to the local shim `compat_getVar` rather than naming an importable library symbol, so no function-pointer stub is needed for `R_getVar` itself. At runtime it always resolves to `findVarInFrame`, since all four call sites in `rpart_callback.c` pass `inherits=FALSE`.

Category E items. The five R interpreter items (`eval`, `findVar`, `findVarInFrame`, `install`, `R_getVar`) are only reachable when `rpart()` is called with `method=4` (user-defined splits). For `install`, the guides specify a self-contained C++ implementation using a `thread_local std::unordered_map<std::string, SEXP>` string-interning cache, which requires no Python registration and is used whenever no callback is registered. The package nevertheless registers a callback for it alongside `eval`, `findVar` and `findVarInFrame`, because the frame registry consulted by `findVarInFrame` is keyed on the pointers interning produces: were the two to come from different tables, every lookup would return `R_UnboundValue`.

`PROTECT`/`UNPROTECT`. Without a garbage collector, the GC protection protocol reduces to no-ops. SEXP lifetime is managed by the Python caller via explicit cleanup after the `.Call` boundary returns.

The header architecture. The guides collectively specify a layered DAG of fake headers, from `fake_arena.h` (lowest level, arena allocator) through type definitions (`INTSXP.h`, `SEXP.h`), accessors (`INTEGER.h`, `REAL.h`), memory management (`R_alloc.h`, `R_Free.h`), I/O (`Rprintf.h`, `error.h`), and finally the Category E function-pointer bridges (`R_getVar.h`, `install.h`, `eval.h`).

5 Phase 4: Fake C++ Header Generation and Compilation Bug Audit

5.1 Motivation

With implementation blueprints established in Phase 3, this phase translates each Markdown guide into actual compilable C++ code. Generating correct headers for 51 interdependent

items is non-trivial: ODR violations, include-guard mismatches, and linkage conflicts are easy to introduce when headers are generated item by item without a global view. The phase therefore includes a systematic post-generation bug audit.

5.2 Header Generation

The `/generate-r-extern-fake-headers` (see Appendix D.1) skill was invoked:

```
/generate-r-extern-fake-headers rpart/src/ r_extern_analysis/combined.csv \
    r_extern_analysis/fake_guides/ r2py_rpart/r_fake_headers/
```

The `@generate-r-extern-fake-header` sub-agent was invoked sequentially for each of the 51 items, producing the corresponding `.h` file in `r2py_rpart/r_fake_headers/`. The generation order preserved the dependency-safe first-occurrence order from the CSV. After all 51 items, the master entry-point header `fake_R.h` was assembled, including all 51 headers in the correct dependency order beginning with `fake_arena.h` as the mandatory first include.

The arena allocator `fake_arena.h` was authored as a prerequisite: it provides `ArenaFrame` (a RAII guard for `.Call` boundaries), `arena_alloc(bytes)` (allocates from the current active frame), and `arena_calloc(n, size)`. Every `.Call` boundary wrapper that calls `R_alloc`/`ALLOC` must declare `ArenaFrame _frame;` as its first local variable; the destructor walks the frame's linked list of allocated blocks and frees them, then restores the previous frame pointer. The allocator is `thread_local` to support the Python ctypes model where each `.Call` invocation runs on a single thread.

5.3 Post-Generation Bug Audit

Following generation, a systematic compilation bug audit was performed. The primary tool was a pair of grep commands over all 53 headers:

```
grep -rh "#define␣FAKE_" *.h | sort -u
grep -rh "#ifndef␣FAKE_" *.h | sort -u
```

Seven compilation bugs were identified and fixed. The root causes fell into three classes: (1) include guard name mismatches (a header that first defines a symbol does not set the guard that a later header checks before emitting its fallback); (2) linkage conflicts (`static inline` fallbacks in companion headers conflicting with `inline` definitions in primary headers); and (3) multi-TU correctness for function-pointer variables.

The most consequential bug was that the four Category E function pointer variables (`g_install_fn`, `g_findVar_fn`, `g_findVarInFrame_fn`, `g_eval_fn`) were declared `static` at namespace scope in their header files. In C++, a `static` variable in a header has internal linkage: every translation unit that includes the header gets its own independent copy. In a multi-translation-unit shared library build, Python's `register_install_fn(fn)` call would update only the copy belonging to the translation unit containing the registration function; all other units would retain `nullptr` and throw an error when the callback path is exercised.

The fix was to change all four from `static` to `inline`, which gives C++17 shared-variable semantics: the linker selects one definition across all translation units.

The complete list of bugs fixed is summarized in Table 1.

Table 1: Compilation bugs identified and fixed during the Phase 4 audit.

Bug	Files Modified	Root Cause and Fix
1	`error.h`	`Rf_warning` defined without setting guard; `warning.h` fallback fired; both definitions in same TU. Fixed by adding `#define FAKE_RF_WARNING_DEFINED` in `error.h`.
2	`R_getVar.h`, `eval.h`	Function pointer variables declared `static`; each TU got its own copy; Python registration only updated one. Fixed by changing to `inline`.
3	`DL_FUNC.h`, two stubs	`DL_FUNC.h` used guard `FAKE_R_BOOLEAN_H`; `Rboolean.h` used `FAKE_RBOOLEAN_DEFINED`; mismatch caused `Rboolean` to be defined twice. Fixed by unifying guard names.
4	`PROTECT.h`	`PROTECT.h` used two separate guards per function; `SEXP.h` used one unified guard; mismatch caused `R_ProtectWithIndex`/`R_Reprotect` to be redefined (`PROTECT_INDEX` is `typedef int`, so types are identical—pure guard error). Fixed by unifying guards in `PROTECT.h`.
5	`error.h`	`error.h` defined `Rf_error` without any guard; `R_NilValue.h` (earlier in include chain) had already defined it with guard set. Fixed by wrapping `error.h`'s definition in the matching guard.

Continued on next page

Bug	Files Modified	Root Cause and Fix
6	`DL_FUNC.h`	Registration functions defined as `inline` in `DL_FUNC.h` without setting their individual guards; companion headers emitted `static inline` fallbacks; `inline`/`static` conflict is ill-formed in C++. Fixed by adding three `#define FAKE_R_*_DEFINED` lines in `DL_FUNC.h`.
7	`DL_FUNC.h`	`DL_FUNC.h` declared `struct DllInfo {}` without setting `FAKE_DLLINFO_DEFINED`; `DllInfo.h` then attempted `typedef struct DllInfo_fake DllInfo`, redefining `DllInfo` as a different type. Fixed by adding the guard definition.

A subtlety confirmed safe, not a bug. The sentinel values `R_NilValue`, `R_UnboundValue`, and `R_NamesSymbol` are each declared as TU-local `static SEXP` variables, initialized by a call to a small `inline` factory function (e.g. `make_nil_value()`) that itself contains a function-local `static SEXPREC` object. Because a `static` local variable inside an `inline` function has exactly one shared instance across all translation units in C++17, every TU's copy of `R_NilValue` ultimately points at the same underlying `SEXPREC`, so pointer-equality comparisons such as `x == R_NilValue` remain correct across the multi-TU build despite the outer variable being `static`. The companion scalar constants `R_NaReal`, `R_NaN`, `R_PosInf`, `R_NegInf`, and `R_NaInt` are compared by value rather than by address, so their independent per-TU `static const` copies are likewise safe.

After all fixes, the 53 headers in `r2py_rpart/r_fake_headers/` are correct with respect to the ODR, guard consistency, linkage consistency, and multi-TU correctness.

6 Phase 5: Pure-C Entry-Point Generation and Fake-Header Migration

6.1 Motivation

With compilable fake headers available, the remaining steps needed before Python can call the C code are: (1) generate wrapper functions that translate between the SEXP-based internal interface and a plain-array interface callable without R, and (2) replace the real R API `#include` directives in the C source files with `#include "fake_R.h"`.

The entry-point wrappers are necessary because the internal C functions take `SEXP` arguments (opaque R list structures) and return `SEXP` values. Python cannot construct SEXP objects directly. The wrappers accept plain `int*`/`double*` arrays (matching how `.Call()` in R pre-converts its arguments) and perform the SEXP construction and deconstruction internally, returning results as plain arrays that Python can read.

6.2 Phase 5A: Entry-Point Generation

The `/generate-r-extern-raw-entry-points` (see Appendix E.1) skill was invoked:

```
/generate-r-extern-raw-entry-points rpart/R/ r2py_rpart/src/ \
    r2py_rpart/r_fake_headers/ r2py_rpart/c_entry_points/
```

The skill first scanned all `.R` files in `rpart/R/` for `.Call()` and `.External()` invocations. Five invocations were found across five files, one per file (`pred.rpart.R`, `rpartcallback.R`, `rpart.exp.R`, `xpred.rpart.R`, `rpart.R`); no `.External()` invocations exist. For each of the five C functions (`pred_rpart`, `init_rpcallback`, `rpartexp2`, `xpred`, `rpart`), the `@generate-r-extern-raw-entry-point` sub-agent generated a corresponding `*_c.c` file in `r2py_rpart/c_entry_points/`. All five wrappers follow a common structural convention:

- `#include "fake_R.h"` is placed before the `extern "C" {` block to avoid C++ template-in-linkage-spec errors.
- `ArenaFrame _frame;` is declared as the first local variable to own all arena allocations made during the call.
- An exception boundary is implemented via `noexcept` plus catch blocks for `RError`, `std::bad_alloc`, `std::exception`, and the catch-all `(...)`, each of which writes a null-terminated error string to the caller-supplied error buffer.

File	Size	Key Design Decisions
`pred_rpart_c.c`	24 KB	12-argument wrapper; `dimx`/`dimc` passed as 2-element `int*`; `csplit2` is nullable (`csplit2_nrow=0` skips SEXP construction); returns `where` as a flat array of length `dimx[0]`.
`init_rpcallback_c.c`	24 KB	`rho`, `expr1`, `expr2` passed as opaque `void*` handles; requires pre-registration of `install` and `findVarInFrame` and a populated frame registry for the four "back" arrays before the call.
`rpartexp2_c.c`	9.8 KB	2-argument wrapper; `dtimes` passed by pointer with no copy; `eps` passed as a scalar `double` via a 1-element buffer; returns a `keep` flag array of length n.
`xpred_c.c`	27 KB	15-argument wrapper; `xmat` kept column-major (R's native layout) so `nrows()`/`ncols()` resolve correctly; `cp` is mutated in place, so the caller must pass a writable copy; returns a flat `double*` that the caller reshapes/transposes on the Python side.
`rpart_c.c`	33 KB	11-argument wrapper, 769 lines, the largest of the five; `parms` falls back to a zero scalar when `NULL`/empty; the return value is a list of 6–7 elements, with `csplit` included only when a `has_csplit` flag is set; a `FREE_INPUTS_AND_RETURN` macro centralizes cleanup on every error path.

The `init_rpcallback_c.c` wrapper is the most architecturally complex: the `rho`, `expr1`, and `expr2` arguments (R environments and expressions) must be passed as opaque `void*` handles, since they represent R interpreter state that cannot be expressed as plain arrays. This is the point where the Python-side frame registry (described in Phase 9) must construct fake SEXP objects wrapping numpy buffers and register the Category E function-pointer callbacks before the C function is called.

6.3 Phase 5B: Bulk Fake-Header Migration

The `/modify-c-folder-with-fake-headers` (see Appendix F.1) skill was invoked:

```
/modify-c-folder-with-fake-headers r2py_rpart/src/ r2py_rpart/c_entry_points/
   r2py_rpart/r_fake_headers/fake_R.h
```

All 35 `@modify-c-file-with-fake-headers` sub-agents were dispatched simultaneously (the modifications are fully independent). Of the 35 files, only 5 required modification:

`rpart.h`, `init.c`, `pred_rpart.c`, `rpart_callback.c`, and `rpartexp2.c`. The remaining 30 files include only package-internal headers and receive their R API access transitively through `rpart.h`, which was itself modified. No R API symbols required suppression (comment-out) in any file—the fake header inventory from Phase 4 covered 100% of the R API surface.

Anonymous struct fix. During compilation verification, 14 files failed with:

```
rpart.h:76:3: error: '<unnamed␣struct>␣rp', declared using unnamed type,
is used but never defined [-fpermissive]
```

`rpart.h` declared the global `rp` struct as an anonymous type, valid in C but rejected by g++ in strict C++ mode. The fix changed `EXTERN struct {` to `EXTERN struct rp_globals_t {`, giving the type a tag name that satisfies C++ linkage requirements. This is the only code change introduced by Phase 5B that was not a mechanical header-include replacement.

After the fix, all 37 translation units (31 package sources + 6 entry-point wrappers) compiled without errors using:

```
g++ -std=c++17 -x c++ -I<fake_headers_dir> -I<c_folder> \
    -Wall -Wno-unused-variable -Wno-unused-parameter -fsyntax-only <file>
```

The 35 files that emitted warnings produced only two pre-existing benign warnings: a multi-line comment in `R_CallMethodDef.h`'s documentation block, and a semantically identical redefinition of `#define _(String) (String)` between `fake_R.h` and `rpart.h`.

7 Phase 6: R Package Structural Dependency Analysis

7.1 Motivation

Having established the C compilation infrastructure in Phases 1–5, the project turns to the R source layer. The 36 R source files define 47 functions; these functions must be converted to Python in a dependency-aware order (callee before caller) and each must be provided with the correct context about its internal and external dependencies. This phase produces the machine-readable structural analysis—JSON dependency maps and a topological level CSV—that drives the systematic conversion in Phase 9.

As in Phase 1's analogous treatment of the C source layer (Section 2), the `/analyze-r-folder-dependencies` (see Appendix G.1) skill dispatches one sub-agent invocation per R file—rather than a single invocation over the whole directory—for the same context-window and auditability reasons: no single invocation can hold all 36 files' source alongside the reasoning needed to classify every call site correctly, and many small, per-file invocations are easier to spot-check than one large one. The classification performed on the R side is finer-grained than its C-side counterpart, however, since R source additionally makes heavy, idiomatic use of base-R and standard-library functions with no C analogue. Each sub-agent classifies every call into exactly one of three categories: *language dependencies* (calls to base R or its standard library, such as `tapply` or `format`), *internal dependencies*

(calls to other functions defined elsewhere in the `rpart` R source), and *external dependencies* (the five `.Call()` invocations that cross into the C back-end). The *internal*/*external* halves of this split are the direct R-side counterpart of Phase 1's two-way classification; *language dependencies* is the one category with no C-side analogue in Phase 1, since it captures R's own idiomatic use of base-R functions, for which the C source has no equivalent concept. This three-way classification is the structural backbone of every subsequent phase: language dependencies drive the Phase 8 conversion-guide catalog, internal dependencies drive the topological conversion order used in Phase 9, and external dependencies confirm the narrow R-to-C interface identified in Section 1.2.

7.2 Methodology

The `/analyze-r-folder-dependencies` (see Appendix G.1) skill was invoked:

```
/analyze-r-folder-dependencies rpart/R/
```

The `@analyze-r-file-dependencies` sub-agent was applied sequentially to each of the 36 R source files. Each agent enumerated all function calls (including calls inside anonymous closures and nested functions) and classified each into one of three categories: *language dependencies* (base R or standard library functions), *internal dependencies* (functions defined in sibling `.R` files), or *external dependencies* (`.Call()`/`.External()` invocations targeting compiled C routines). Results were saved to `structural_analysis/R/<stem>.json`.

The pre-existing `dependency_graph.py` and `dependency_levels.py` scripts were then run to produce the graph visualization and the topological level CSV `structural_analysis/dependency_levels.csv`.

7.3 Key Findings

The full call graph contains 262 nodes (36 file nodes, 49 function nodes, 172 language-dependency nodes, 5 external-dependency nodes) and 745 edges. Only 5 of the 36 files contain `.Call()` invocations, confirming that the R-to-C interface is narrow.

Level	Count	Description
0	20	Entry points; no other internal function calls these
1	18	Called by level-0 functions
2	4	Called by level-1 functions
3	1	Called by level-2 functions
4	2	Called by level-3 functions
5	2	Deepest leaves in the call DAG

The 47 functions are stratified into these 6 dependency levels, of which 28 make no internal calls themselves. The two deepest utilities are `compress` (level 5, the subtree-overlap elimination algorithm inside `rpartco`) and `tree.depth` (level 5, computes node depth from binary node number via `floor(log2(node))`). The most widely referenced internal functions are

`formatg` (called by all four method initializers and `labels.rpart`), `na.rpart` (called by `model.frame.rpart`, `rpart`, and `xpred.rpart`), and `rpart.control`, `rpart.matrix`, and `rpartcallback` (each called by both `rpart` and `xpred.rpart`).

The `rpart` and `xpred.rpart` functions are the two primary orchestrators, each pulling in `na.rpart`, `rpart.matrix`, `rpartcallback`, and all four method-dispatch targets dynamically via `get(paste("rpart", method, sep="."))`. `zzz.R` defines five package-utility functions (`tree.depth`, `string.bounding.box`, `node.match`, `descendants`, `.onUnload`) that are consumed by the visualization layer (`rpartco.R`, `text.rpart.R`, `path.rpart.R`) but carry no internal dependencies of their own.

8 Phase 7: Python Build Infrastructure

8.1 Motivation

The C compilation infrastructure from Phases 4–5 must be integrated into a proper Python packaging workflow before any Python-level testing or distribution is possible. This phase establishes a working `pip install` flow for the `r2py_rpart` package and designs the CI/CD pipeline for multi-platform wheel publication on PyPI.

Two related but distinct objectives determine the scope of this phase. The first is local: the compiled extension must build and install correctly in the HPC development environment so that the Python wrapper functions written in later phases can be developed and tested interactively. The second is distributional: once the package reaches a stable state, it must be available as pre-compiled binary wheels on PyPI for every major platform, so that end users can install it with a plain `pip install r2py_rpart` without needing a C++ compiler or any build toolchain of their own. Because the compiled extension is platform- and Python-version-specific, a separate wheel must be built for every combination of operating system, processor architecture, and CPython version targeted.

The reference design for the CI/CD infrastructure is the analogous project `r2py_kernsmooth` at `python-KernSmooth/r2py_kernsmooth/`, which builds and publishes wheels for a sibling R-to-Python conversion of the `KernSmooth` package. Although the two packages share a publication pipeline, their build architectures differ in every other respect:

Aspect	`r2py_kernsmooth`	`r2py_rpart`
Source language	Fortran	C (compiled as C++)
Binding tool	f2py (NumPy built-in)	cffi (ABI mode, `dlopen`)
Meson build target	`py.extension_module()`	`shared_library()`
Python import	`from . import _KernSmooth`	`ffi.dlopen(path)`
External system dependency	OpenBLAS	none
macOS linker flag	automatic	`-lc++` (explicit)
Windows toolchain	GCC + Fortran + OpenBLAS	GCC only

8.2 Build System Bugs and Fixes

Bug 1: C++17 math built-in undefined references. `meson.build` originally compiled all sources with `-std=c++17`. On the project's GCC 11/glibc system, C++17 mode causes `<cmath>` to declare `std::iscanonical`, `std::issignaling`, `std::iseqsig`, and related overloads that reference GCC built-ins absent from this system's glibc. A manual test confirmed that `-std=c++14` produces zero undefined references. The fix was to change `default_options: ['cpp_std=c++17']` to `['cpp_std=c++14']` and update the `c_args` accordingly. The `inline` variable declarations in the fake headers (a C++17 feature) continue to compile under GCC 11 in C++14 mode as an accepted language extension.

Bug 2: Math library not linked. After the C++14 fix, `log`, `sqrt`, `pow`, and related standard math functions produced undefined references at link time. In C++14 mode these resolve to real library calls requiring `-lm`. The fix added `link_args: ['-lstdc++', '-lm']` to `meson.build`.

Bug 3: `_find_lib()` searched the source tree. The `_find_lib()` function in `__init__.py` computed its search path from `__file__`, which points into the source directory when pytest runs from the source tree rather than a site-packages install. The fix extended the function with a two-stage search: first the package directory co-located with `__file__`, then the `platlib`/`purelib` site-packages directories obtained via `sysconfig.get_path()`.

Bug 4: `_check_error()` received a numpy array. All public wrapper functions passed the error buffer as a numpy `uint8` array to `_check_error()`, which called `ffi.string()` on it. `cffi`'s `ffi.string()` requires a `_CDataBase` object. The fix added a branch that handles `numpy.ndarray` input by splitting the byte sequence at the first null terminator.

8.3 CI/CD Pipeline

A GitHub Actions workflow `.github/workflows/python-publish.yml` was designed to build wheels for five platforms: Linux x86_64, Linux aarch64, macOS Intel (x86_64), macOS Apple Silicon (arm64), and Windows x86_64. The workflow uses `cibuildwheel` and is triggered by a GitHub Release publication event. It mirrors the KernSmooth structure with two adaptations. First, rpart requires no external system library (no BLAS, no Fortran runtime), so the Linux and macOS `before-all` hooks that install openblas are omitted. Second, on macOS, Apple clang links against `libc++` rather than GCC's `libstdc++`; `meson.build` was updated with a conditional:

```
if host_machine.system() == 'darwin'
  cxx_stdlib_link = ['-lc++']
else
  cxx_stdlib_link = ['-lstdc++']
endif
```

After all four bug fixes, the package builds successfully with `pip install -no-build-isolation .` and the existing `rpartexp2` tests pass: `test_rpartexp2_basic`

(input `[1.0, 2.0, 2.0, 3.0, 3.0, 3.0, 5.0]`, asserting a keep/drop array of matching length with values in $\{0, 1\}$) and `test_rpartexp2_all_unique` (input `[1.0, 2.0, 4.0, 8.0]`, asserting that every element is kept). The resulting wheel, `r2py_rpart-0.1.0-cp314-cp314-linux_x86_64.whl` (179,418 bytes), installs `_rpart_core.so` into site-packages.

9 Phase 8: Language Dependency Analysis and Conversion Guide Generation

9.1 Phase 8.1: Language Dependency Extraction

9.1.1 Motivation

Converting an R function to Python requires not just knowledge of the function's own logic but also precise documentation of how each base-R language construct it uses should be translated. Base-R functions like `tapply`, `unlist`, `match`, `lapply`, and `model.frame` have non-trivial semantics that do not map one-to-one to any single Python or NumPy operation. Before conversion guides can be generated for these constructs, their usage locations across the entire codebase must be inventoried to identify all structurally distinct call patterns.

One may object that these translation nuances are well understood and could instead be resolved case by case as each function is converted in Phase 9. This is not a safe simplification. Because each function-conversion agent operates as an independent invocation with no persistent memory of decisions made for other functions, the stochastic nature of LLM inference means that the same construct can be translated correctly in one function and subtly incorrectly in another, with no mechanism to detect the inconsistency short of exhaustive pairwise comparison. Some constructs compound this risk further: `as.integer`, for instance, is called 33 times across the codebase in at least half a dozen structurally distinct contexts (bare scalars, vectors, matrix dimensions, `.Call` argument packing), and a translation strategy correct for one context is not automatically correct for another. Cataloging every call site of every language dependency *before* any function conversion begins, and generating one authoritative, per-construct Markdown guide covering every distinct usage pattern observed in the codebase, converts an implicit, per-agent judgment call into an explicit, auditable, and—critically—*reusable* artifact: the same guide is handed to every conversion agent that happens to use that construct, guaranteeing a single, consistent translation strategy package-wide and giving a human reviewer a single, well-defined location to correct any inaccuracy before it can propagate into multiple converted functions.

This guides-before-generation strategy mirrors the one already used on the C side in Phase 3 (Section 4), where a full implementation blueprint was written for each fake-header item before any header code was generated, precisely to avoid the ODR violations and linkage conflicts that would otherwise arise from independently generated, mutually inconsistent headers. The specific risk differs—semantic drift between independently generated Python translations here, versus binary linkage conflicts between independently generated C++ headers there—but the underlying design principle is the same: settle every cross-cutting

decision once, in a single reusable artifact, before parallel or independent generation begins, rather than leaving it to be rediscovered, possibly inconsistently, by each downstream agent invocation.

9.1.2 Methodology

The `/extract-r-folder-language-dependencies` (see Appendix H.1) skill was invoked:

```
/extract-r-folder-language-dependencies rpart/R/ structural_analysis/R/
```

Each of the 36 R files was processed by the `@extract-r-file-language-dependencies` sub-agent, which was provided both the R source file and its corresponding structural analysis JSON from `structural_analysis/R/`. The agent identified all invocation sites of the language dependencies cataloged in the JSON, extracted the line number and call body for each, and wrote a 4-column CSV to `language_dependency_analysis/R/`. The 36 CSVs were merged by the script `language_dependency_analysis/combine_csvs.py` into `combined_table.csv`.

9.1.3 CSV Encoding Defect

The initial merge failed with a `pandas.errors.ParserError`. A diagnostic loop identified `rpart.anova.csv` as the offending file: `call_body` fields containing R string literals (which use `"` as delimiters) were written with bare `"` characters rather than the RFC 4180-compliant doubled `""` escape sequence, causing the CSV parser to split fields at interior double-quotes. The file was corrected by rewriting it with Python's `csv.writer` using `quoting=csv.QUOTE_MINIMAL`.

9.1.4 Key Findings

The combined table contains 1,298 invocation records across 172 distinct language dependencies. The most frequently called base-R functions are `stop` (80 call sites), `c` (67), `length` (63), `is.null` (60), `cat` (41), `match` (35), `list` (34), `names` (34), `as.integer` (33), `format` (32), `matrix` (30), `missing` (30), `any` (27), `attr` (27), and `ncol` (26).

File	Rows
`rpart.R`	171
`xpred.rpart.R`	98
`rpart.class.R`	87
`summary.rpart.R`	86
`rpart.exp.R`	74

The `rpartcallback.R` file is the densest single-function file: 84 rows reflecting heavy use of `stop` (14 calls), `length` (13 calls), and `assign` (9 calls) for environment construction.

9.2 Phase 8.2: Conversion Guide Generation

9.2.1 Motivation

With the full inventory of 172 language dependencies and their usage contexts available, conversion guides can be generated. Each guide documents the R semantics of one language construct, identifies all structurally distinct call patterns found in the rpart codebase, and provides concrete Python (NumPy/pandas/stdlib) equivalents with notes on type coercion, indexing conventions, and edge cases. These guides serve as the primary reference for the `@convert-r-function-to-python` agents in Phase 9.

9.2.2 Methodology

The `/generate-language-dependency-conversion-guides` (see Appendix I.1) skill was invoked:

```
/generate-language-dependency-conversion-guides rpart/R/
   language_dependency_analysis/combined_table.csv
```

The 172 unique `language_dependency` values were extracted from the combined table using Python's `csv.DictReader` (required because `awk` mishandles multi-line quoted fields). Per-dependency CSV subsets were written to a session scratchpad. Dependency names containing filesystem-unsafe characters were encoded with a custom scheme (e.g., `names<-` → `names_LT_-.md`). The 172 `@generate-language-dependency-conversion-guide` agents were dispatched across 9 parallel batches of up to 20 agents each, all as background tasks. Output guides were written to `language_dependency_analysis/conversion_guides/`. Parallel dispatch is appropriate here, in contrast to the sequential dispatch used in Phase 6, because the guide for each language dependency is logically independent of every other guide: there is no ordering constraint between them, and parallelism reduces wall-clock time from a cost proportional to 172 sequential invocations to a cost proportional to 9 sequential batches. This independence-implies-parallelism reasoning is the same one already applied on the C side in Phase 5B, where all 35 fake-header-migration sub-agents were dispatched simultaneously because the per-file header-include replacements are likewise fully independent of one another; it stands in direct contrast to Phase 3's fake-header *guide* generation and Phase 9's function conversion, both of which are sequential precisely because later items there do depend on decisions made for earlier ones.

Each guide follows a fixed four-part structure: an overview of the construct's functionality and translation challenges; a contextual usage analysis enumerating every call site in the `rpart` codebase grouped by usage pattern; a Python conversion strategy describing the general translation approach and any necessary workarounds; and step-by-step conversion examples pairing the original R code with its Python translation for each distinct usage pattern identified. Agents were permitted to consult R's and Python's online documentation when a construct's behavior was complex or non-obvious.

9.2.3 Verification Edge Case

Two guides are dot-prefixed (`.getXlevels.md` and `.checkMFClasses.md`) and therefore invisible to shell globbing (`ls *.md` or `glob('*.md')`). Final verification required `os.listdir` or `find` to obtain the correct count of 172. An artifact directory `.tmp_csv/` was also present and correctly excluded from the count.

9.2.4 Representative Guide Complexity

The guides cover a wide range of translation complexity. Simple constructs like `any` map directly to `np.any()`. Complex constructs require pattern-specific guidance: `tapply` requires three distinct translations depending on whether the grouping is a free-form factor, a factor with explicit levels for zero-filling, or dense integer bin indices. `t` (transpose) requires distinguishing between a pure orientation transpose (`.T` numpy view) and a transpose-for-C-call that must be followed by `.ravel()` to produce a column-major flat array. `unlist` requires four distinct patterns depending on whether the input is a list of arrays, a list with NULL slots, a column-major DataFrame, or a named scalar/array packing for a `.Call` argument. The `stop` guide alone documents eight distinct call-site scenarios across its 79 invocations—static string messages, `gettextf`-interpolated messages, object-class guards, input-shape and value guards, environment-state guards, and callback-validation guards—reflecting how heavily the package relies on this single function for input validation. The `switch` guide documents three patterns: string-to-string mapping (a plain `dict` lookup), side-effect block dispatch (an `if`/`elif` chain), and vectorised numeric dispatch (an `if`/`elif` chain built around `np.log`, `np.sqrt`, and `np.where`). The `x$functions$text` guide documents rpart's S3 named-callable dispatch idiom, translated to a two-level Python `dict`-of-callables access pattern, `x["functions"]["text"](...)`.

10 Phase 9: R-to-Python Function Conversion and Package Assembly

10.1 Phase 9.1: Batch R-to-Python Function Conversion

10.1.1 Motivation

With all prerequisite artifacts in place—structural dependency maps (Phase 6), language dependency conversion guides (Phase 8.2), and a working C back-end (Phases 4–5)—the systematic translation of the 36 R source files and 47 functions to Python can begin. The conversion must respect the topological order established by the dependency graph: functions whose output is consumed by other functions must be converted first.

A basic design decision precedes the conversion itself: should the unit of translation be an entire R source file, or a single function? Phases 1, 2, 6, and 8 all process entire files (or, in Phase 8's case, one language dependency's full call-site subset) per agent invocation, since a file is naturally self-contained and its context remains manageable even when it defines several functions. Converting the package to Python, however, requires each agent to receive

not only the R source being translated but also its dependency map, the relevant subset of the 172 language-dependency conversion guides, and—critically—the already-converted Python signatures of every internal callee, so that call expressions to sibling functions are constructed correctly. For a multi-function file such as `rpart.R` or `xpred.rpart.R`, bundling this context at file granularity would force every function in the file to be translated in one shot, discarding the topological ordering established in Phase 6 and burying each function's relevant guide subset inside a much larger, less focused prompt. Converting one function per agent invocation instead keeps each conversion unit small and well defined, allows the dependency-level ordering to be enforced exactly—even across functions that happen to share a source file—and lets each agent's context consist only of what is relevant to the function at hand.

10.1.2 Methodology

The `/convert-r-folder-to-python` (see Appendix J.1) skill was invoked:

```
/convert-r-folder-to-python rpart/R/ structural_analysis/R/
    structural_analysis/dependency_levels.csv
    language_dependency_analysis/conversion_guides/ conversion_results/R/
```

The skill parsed the dependency level CSV and sorted the 36 files in descending order of their maximum dependency level, ensuring that deep callees were converted before their callers. Within each multi-function file, functions were further sorted leaf-to-root. For example, `rpartco.R` was processed with `compress` (level 5) before `rpartco` (level 4); `rpart.exp.R` with `drate2` (level 2) before `rpart.exp` (level 1). Independent leaf functions across different files were dispatched in parallel.

Each `@convert-r-function-to-python` sub-agent was provided the R source file, the structural dependency JSON, the full set of 172 conversion guides, and the dependency level CSV. The output for each function was a JSON file with keys `imports`, `function_prototype`, and `function_body`, saved to `conversion_results/R/<stem>/<function>.json`.

10.1.3 Cross-Cutting Translation Decisions

The following R-to-Python idiom mappings were applied uniformly across all conversions:

R Construct	Python Equivalent
R 1-based indexing	Python 0-based indexing throughout
`list(a=1, b=2)` (named list)	`dict`
`missing(arg)`	Module-level `_MISSING = object()` sentinel
`match(x, table, 0L)`	Dict-based lookup; 0 or -1 for not found
`rep(x, n)`	`[x] * n` or `np.repeat`
`rbind`/`cbind`	`np.vstack`/`np.column_stack` or `np.hstack`
`cumsum(c(...))`	`np.cumsum(np.concatenate([...]))`
`pmax`/`pmin`	`np.maximum`/`np.minimum`
`tapply(wt, factor(y), sum)`	`pd.Series.groupby(pd.Categorical).sum()`
`factor`/`levels`	`pd.Categorical`
`outer(x, y, FUN)`	NumPy broadcasting
`apply(m, 1, paste, collapse=sep)`	List comprehension with `str.join` over rows
`attr(x, "terms")`	`x.attrs.get("terms")`
Column-major matrix for `.Call`	`np.reshape(..., order='F')`
`t(Y)` before `.Call`	`Y.T.ravel()`
`.Call("C_rpart")`	`from r2py_rpart import _rpart_c`
`new.env()`/`assign()`/`get()`	Python `dict`
`asNamespace("rpart") + environment<-`	No-op; omitted
`UseMethod(...)`	Dispatch stub with `NotImplementedError`
R graphics (`plot`, `polygon`, ...)	matplotlib equivalents

10.1.4 Notable Per-File Decisions

Several functions required non-mechanical decisions. `compress` in `rpartco.R` is a self-recursive nested function; it was extracted as a standalone module-level function with the closure variables (`is_leaf`, `nspace`) passed as explicit parameters. Similarly, `drate2` (nested inside `rpart.exp`) and `tfun` (nested inside `rpart`) were extracted as standalone helpers and also redefined inline in their enclosing function's conversion.

The largest function, `rpart()` (300 lines), required careful handling of the ordered-factor categorical split matrix construction, classification probability vector normalization, and assembly of the final output dictionary. R's S3 class assignment `class(ans) <- "rpart"` was replaced with a sentinel entry `ans["_rpart_class"] = "rpart"`, and `attr()`-stored metadata was handled via `ans["_xlevels"]` and `ans["_ylevels"]`.

`oval` and `rectangle` in `text.rpart.R` translate R's `polygon(x, y)` calls to matplotlib's `ax.fill(x, y)` (a filled polygon drawn against an `Axes` object), and `post.rpart`'s PostScript tree export—built on R's `postscript()` graphics device—was translated using matplotlib's `FigureCanvasPS` backend.

The `eval.parent(model.frame(...))` pattern—which dynamically constructs a model frame from the parent call object—was identified as a hard boundary that cannot be faithfully reproduced in Python without a formula parser. All such sites were stubbed with

`NotImplementedError`, to be filled in once patsy integration is available. The interactive `snip.rpart.mouse` function was also stubbed, as terminal interaction is not reproducible in Python.

10.2 Phase 9.2: Package Assembly, API Restructuring, and Test Validation

10.2.1 Motivation

The 47 JSON function definitions produced in Phase 9.1 exist as isolated per-function artifacts. This phase assembles them into importable Python modules, resolves inter-module imports, establishes the package's public API, and validates the assembled package against a test suite.

10.2.2 Package Assembly

The `/combine-python-functions-into-folder` (see Appendix K.1) skill was invoked:

```
/combine-python-functions-into-folder conversion_results/R/
   r2py_rpart/r2py_rpart/ r2py_rpart/r2py_rpart/
```

A Python script read each conversion subdirectory's JSON files, deduplicated imports, prepended the `from __future__ import annotations` and `from typing import Any` boilerplate, and assembled functions as prototype-plus-body blocks. All 36 files were written successfully. Twenty-six of the 36 output filenames contained dots from R naming conventions (e.g., `labels.rpart.py`) and were renamed by replacing internal dots with underscores (e.g., `labels_rpart.py`) to make them importable as Python modules. The full renaming map:

Original	Renamed
`labels.rpart.py`	`labels_rpart.py`
`meanvar.rpart.py`	`meanvar_rpart.py`
`model.frame.rpart.py`	`model_frame_rpart.py`
`na.rpart.py`	`na_rpart.py`
`path.rpart.py`	`path_rpart.py`
`plot.rpart.py`	`plot_rpart.py`
`post.rpart.py`	`post_rpart.py`
`pred.rpart.py`	`pred_rpart.py`
`predict.rpart.py`	`predict_rpart.py`
`print.rpart.py`	`print_rpart.py`
`prune.rpart.py`	`prune_rpart.py`
`residuals.rpart.py`	`residuals_rpart.py`
`roc.rpart.py`	`roc_rpart.py`
`rpart.anova.py`	`rpart_anova.py`
`rpart.branch.py`	`rpart_branch.py`
`rpart.class.py`	`rpart_class.py`
`rpart.control.py`	`rpart_control.py`

Original	Renamed
rpart.exp.py	rpart_exp.py
rpart.matrix.py	rpart_matrix.py
rpart.poisson.py	rpart_poisson.py
rsq.rpart.py	rsq_rpart.py
snip.rpart.mouse.py	snip_rpart_mouse.py
snip.rpart.py	snip_rpart.py
summary.rpart.py	summary_rpart.py
text.rpart.py	text_rpart.py
xpred.rpart.py	xpred_rpart.py

Four categories of import-time errors were identified and fixed before tests could run:

1. **Undefined `_MISSING` sentinel** in four files: `rpart.py`, `rpart_control.py`, `summary_rpart.py`, `xpred_rpart.py`. The `_MISSING` sentinel is used as a default argument value (evaluated at function definition time); it was defined in some files but absent in others. Fixed by adding `_MISSING = object()` before each affected function.
2. **Stale absolute imports** in `rpart_exp.py` and `rpart_poisson.py`: the conversion agent emitted paths of the form `from conversion_results.R.formatg.R.formatg import formatg`, which are unreachable at runtime. Fixed by replacing with relative imports: `from .formatg import formatg`.
3. **Module-level `import patsy` in `rpart_matrix.py`**: `patsy` is not installed in the target environment. The import was moved inside the function body as a lazy import.
4. **Missing sibling-module imports in `rpart.py`**: the converted `rpart()` function called `rpart_anova`, `rpart_class`, `rpart_control`, `rpart_exp`, `rpart_matrix`, `rpart_poisson`, `rpartcallback`, and `importance` without importing them (R's namespace made them available automatically). Eight relative imports were added.

10.2.3 API Restructuring

The original `__init__.py` exposed the four C-level wrapper functions as `rpart`, `pred_rpart`, `xpred`, and `rpartexp2`. After assembly, the high-level Python interfaces from the converted modules claimed the same names. The C wrappers were renamed to private symbols (`_rpart_c`, `_pred_rpart_c`, `_xpred_c`, `_rpartexp2_c`), and the high-level Python functions were promoted to the public API via `__all__`. The three modules that previously imported the C wrappers by their old public names were updated accordingly. After restructuring, `r2py_rpart.__all__` contains 49 entries; the primary user-facing exports are:

Symbol	Source Module	R Equivalent
rpart	rpart.py	rpart::rpart()
pred_rpart	pred_rpart.py	rpart:::pred.rpart()
predict_rpart	predict_rpart.py	predict.rpart()
xpred_rpart	xpred_rpart.py	rpart:::xpred.rpart()
rpart_control	rpart_control.py	rpart::rpart.control()
summary_rpart	summary_rpart.py	summary.rpart()
plot_rpart	plot_rpart.py	plot.rpart()
text_rpart	text_rpart.py	text.rpart()
prune_rpart	prune_rpart.py	prune.rpart()
importance	importance.py	rpart:::importance()

C wrappers retained as private symbols: _rpart_c, _pred_rpart_c, _xpred_c, _rpartexp2_c.

10.2.4 Runtime Bugs in rpart.py

Three bugs in the assembled rpart.py were discovered and fixed during test development:

1. **Off-by-one errors in isplit[:, 0] variable indices.** The C entry point rpart_c returns isplit[:, 0] as 1-based variable indices (R convention), where index 0 is the sentinel for leaf nodes and index $k \geq 1$ refers to the k-th predictor. The conversion had incorrectly treated these indices as 0-based in three locations: the split row-name construction, the isord array lookup, and the frame["var"] column. All three were corrected by subtracting 1 before 0-based array accesses.

2. **parms dict serialization for classification method.** rpart_class returns init["parms"] as a dict with keys prior, loss, and split. The original serialization attempted np.array(init["parms"]).ravel(), which produces a 0-dimensional object array on a dict and raises TypeError. The fix iterates over .values() and concatenates the flattened arrays.

3. **Circular import avoidance.** rpart.py imports _rpart_c lazily inside the function body (not at module level) to avoid a circular import: __init__.py imports rpart from rpart.py at module level; a module-level import of _rpart_c from r2py_rpart would trigger __init__.py to run again before it has finished loading.

10.2.5 Build System Fix

After reinstalling with pip install -no-build-isolation ., the test runner reported ModuleNotFoundError: No module named 'r2py_rpart.formatg'. Inspection showed that only __init__.py was present in site-packages. The root cause is that meson-python requires every .py file to be enumerated explicitly in py.install_sources(); it does not auto-discover Python source files. The original listing had only one entry. The fix added all 36 new modules, bringing the total to 37 entries. This is a structural property of meson-python

that differs from setuptools (`find_packages()`) and must be maintained: any session that adds new Python modules must also update `meson.build` in the same change.

10.2.6 Test Results

A test suite of 11 tests for the primary `rpart()` function was written and all 11 pass. The tests cover: output dictionary structure, `frame` type, `cptable` column names, `where` vector semantics, method recording for both `anova` and `class` methods, splitting behavior on a synthetic signal-vs-constant dataset, and error handling for invalid inputs.

10.3 Phase 9.3: R Interpreter Item Bridge and `cffi` Migration

10.3.1 Motivation

Two gaps remained after Phase 9.2 before the package could support the user-defined split path (`method=4`). First, the C side was missing five helper functions needed by the Python-side frame registry to construct fake SEXP objects wrapping numpy buffers (`make_real_sexp`, `make_int_sexp`, `make_env_sexp`, `free_sexp_helper`, and access to `get_make_sexp_error`). Second, the converted `rpartcallback.py` used the `ctypes` module for callback registration and SEXP construction, which is incompatible with the package's `cffi`-based `_lib` object.

10.3.2 Methodology

The `/add-r-interpreter-items` (see Appendix L.1) skill was invoked:

```
/add-r-interpreter-items r_extern_analysis/fake_guides/
   r2py_rpart/c_entry_points/ r2py_rpart/r2py_rpart/ r2py_rpart/
```

The skill identified three Category E guides whose opening blockquote begins with **R Interpreter Item.** (strict prefix match): `eval.md`, `findVar.md`, and `findVarInFrame.md`. The `install.md` guide was excluded because its opening blockquote notes that the `install` function is best-effort fakeable in C++ without a Python function pointer—its built-in hash-map implementation in `install.h` is sufficient for normal use.

The `@add-r-interpreter-items` agent was invoked once with the full content of all relevant files, including the fake guides, the existing `init_rpcallback_c.c` wrapper (authoritative protocol reference), `fake_R.h`, `__init__.py`, and `rpartcallback.py`.

10.3.3 C Helper File

A new file `r2py_rpart/c_entry_points/interpreter_helpers_c.c` was written exporting five `extern "C" noexcept` functions:

Function	Signature	Purpose
`make_real_sexp`	`void*(void*, int)`	Allocate REALSXP node; data not copied
`make_int_sexp`	`void*(void*, int)`	Allocate INTSXP node; data not copied
`make_env_sexp`	`void*(void)`	Allocate minimal ENVSXP identity shell
`free_sexp_helper`	`void(void*)`	Free SEXPREC node only (not data buffer)
`get_make_sexp_error`	`const char*(void)`	Return thread-local error string

The five helpers operate directly on the `SEXPREC` layout established in Phase 4's `INTSXP.h`:

```
struct SEXPREC { int type; int length; int nrow; int ncol; void *
   data; };
typedef SEXPREC *SEXP;
```

using the constants `REALSXP=14`, `INTSXP=13`, and `ENVSXP=4`.

The function `call_install` was *not* added as a new C-level function: `install.h` already emits it as a weak (`W`) `extern "C" inline` symbol via `-fkeep-inline-functions`, and redefining it in a `.c` file would cause a duplicate-symbol linker error. The correct approach is to declare it in `ffi.cdef()` only, reusing the pre-existing weak symbol.

10.3.4 rpartcallback.py Rewrite

The full rewrite replaced every `ctypes` FFI call with its `cffi` equivalent:

Old (`ctypes`)	New (`cffi`)
`ctypes.CFUNCTYPE(...)`	`ffi.callback("...", fn)`
`class _SEXPREC(ctypes.Structure): ...`	`_lib.make_real_sexp(buf, size)`
`(ctypes.c_char * 1)()`	`ffi.new("char[1]")`
`ctypes.cast(node, ctypes.c_void_p).value`	`int(ffi.cast("uintptr_t", node))`
`arr.ctypes.data_as(ctypes.c_void_p)`	`ffi.from_buffer(arr)`
`_lib.register_*_fn.restype = ...`	`_lib.register_*_fn(ffi.cast("void *", cb))`

A subtle lifetime consideration: `cffi`'s `ffi.from_buffer(arr)` returns a cdata object that pins the numpy buffer for the duration of the call, but both the buffer object and the `from_buffer` result must be kept in the `_cffi_keep_alive` list to prevent premature garbage collection. The SEXP node pointers allocated by `make_real_sexp` for the `yback`/`wback`/`xback`/`nback` buffers are kept conservatively in the same list, even though after `init_rpcallback_c` returns the C static globals point directly into the numpy buffers, not into the SEXP nodes.

A distinction that was explicitly preserved: the three helper functions `_iptr`, `_dptr`, `_cptr` in `__init__.py` call `arr.ctypes.data` (numpy's `.ctypes` attribute, which returns an integer memory address) and pass it to `ffi.cast(...)`. This is valid `cffi` usage—`numpy.ndarray.ctypes` is a numpy feature unrelated to the Python `ctypes` module—and was correctly retained without modification.

10.3.5 Results

After the build (`pip install -no-build-isolation .`), symbol verification confirmed the presence of all expected symbols in `_rpart_core.so`: `make_real_sexp`, `make_int_sexp`, `make_env_sexp`, `free_sexp_helper` as strong (`T`) symbols from `interpreter_helpers_c.c`, and `call_install` as a weak (`W`) symbol from `install.h`.

Symbol	Type	Source
`make_real_sexp`	T (strong)	`interpreter_helpers_c.c`
`make_int_sexp`	T (strong)	`interpreter_helpers_c.c`
`make_env_sexp`	T (strong)	`interpreter_helpers_c.c`
`free_sexp_helper`	T (strong)	`interpreter_helpers_c.c`
`call_install`	W (weak)	`install.h` (inline)
`register_eval_fn`	T	`eval.h`
`register_install_fn`	T	`R_getVar.h`
`register_findVar_fn`	T	`R_getVar.h`
`register_findVarInFrame_fn`	T	`R_getVar.h`
`get_R_UnboundValue`	T	`R_UnboundValue.h`

All 11 regression tests continue to pass. The `method=4` user-defined splits code path is architecturally complete at the FFI layer; end-to-end testing with actual user-defined split functions has not yet been performed.

11 Phase 10: Full Equivalence Audit and Callback Bridge Repair

11.1 Motivation

Phase 9 closed with a package that built, installed, and passed an 11-test regression suite exercising the primary `rpart()` entry point on two synthetic datasets. Passing a small, hand-written suite is a necessary but not sufficient condition for behavioral equivalence with R: the public surface of `rpart` spans twenty-three documented functions (per the systematic `rpart-internal.Rd` cross-reference formalized in Phase 12.1) covering fitting, prediction, pruning, cross-validation, printing, and four distinct plotting routines, and the Phase 9.1 conversion agents worked function-by-function from R source and 172 language-construct guides without ever running the fully assembled package end-to-end against a live R session. Before any further work is layered on top of the port, this phase performs a systematic, file-by-file audit of every converted module against its R counterpart, and simultaneously

closes out the single documented outstanding limitation carried over from Phase 9.3: the `method=4` (user-defined split) code path, which was architecturally wired at the FFI layer but had never been exercised with an actual user-supplied splitting function.

11.2 A Six-Way Parallel Audit

Rather than one agent reading all 36 R/Python file pairs sequentially, the audit was partitioned by subsystem and dispatched as six `general-purpose` sub-agents running in parallel—Claude Code's built-in, project-agnostic agent type, used here in place of a bespoke project skill/agent pair of the kind employed in Phases 1–9, because exhaustive line-by-line comparison of an R file against its Python translation is a generic reading-and-reasoning task rather than a repeatable pipeline step that would benefit from being encoded as a reusable command. Each agent was given the R sources, the corresponding Python modules, and any relevant C sources for one subsystem, and was instructed to report discrepancies by file and line number together with a severity rating.

Subsystem	Files Audited
Core fit pipeline	`rpart.py`/`rpart.R`, `rpart_matrix.py`, `na_rpart.py`, `model_frame_rpart.py`, `rpart_control.py`, the cffi bridge in `__init__.py`, `c_entry_points/rpart_c.c` vs. `src/rpart.c`
Callback/method dispatch	`rpartcallback.py`, `rpart_anova.py`, `rpart_poisson.py`, `rpart_class.py`, `rpart_exp.py`, `c_entry_points/init_rpcallback_c.c`, `c_entry_points/interpreter_helpers_c.c`
Prediction	`predict_rpart.py`, `pred_rpart.py`, `c_entry_points/pred_rpart_c.c`, `src/rundown.c`/`rundown2.c`
Pruning/CP/cross-validation	`prune.py`, `prune_rpart.py`, `printcp.py`, `plotcp.py`, `xpred_rpart.py`, `snip_rpart.py`, `snip_rpart_mouse.py`, `c_entry_points/xpred_c.c`
Display/plotting	`print_rpart.py`, `plot_rpart.py`, `text_rpart.py`, `post.py`/`post_rpart.py`, `rpartco.py`, `labels_rpart.py`, `rpart_branch.py`
Misc. statistical utilities	`formatg.py`, `importance.py`, `meanvar_rpart.py`, `path_rpart.py`, `residuals_rpart.py`, `roc_rpart.py`, `rsq_rpart.py`, `summary_rpart.py`, `zzz.py`

One agent stalled mid-run on the pruning/cross-validation cluster and was resumed via `SendMessage` rather than restarted. Findings were spot-checked before acceptance—most notably, a claimed column-major/row-major corruption in the cffi marshaling layer was independently reproduced with a standalone NumPy byte-comparison script—and the six reports were synthesized into one consolidated, severity-ranked findings document, which was presented to the user for approval before any source file was modified.

11.3 Comprehensive Bug-Fix Pass

Following approval, every finding was fixed directly in the Python sources under a `TodoWrite`-tracked checklist, with the package rebuilt (`pip install -no-build-isolation .`) and the test suite re-run after each individual change, per the user's explicit constraint that `r2py_rpart/src/`—the literal, compiled-from-R C sources—must never be modified. Selected fixes, in rough order of severity, are summarized in Table 3.

Table 3: Selected bugs fixed during the Phase 10 equivalence audit.

File(s)	Defect and Fix
`__init__.py`	`np.ascontiguousarray(arr, dtype=...)` inside `_int32`/`_float64` silently flattened Fortran-ordered 2-D inputs back to C order, corrupting every multi-row, multi-column matrix (`xmat`, `nodes2`, `split2`, `csplit2`, `xdata2`, `xmiss2`) passed to the C core; replaced with an order-preserving `np.asarray(..., order="K")` plus an explicit contiguity fallback, and added explicit buffer-capacity checks before every output reshape.
`rpart.py`, `model_frame_rpart.py`	The documented `rpart(formula, data=df)` call pattern raised `NotImplementedError` outright; implemented a from-scratch, regex-based R-style formula parser (`"y ~ x1 + x2"`, `"y ~ ."`) and the main branch of `model_frame_rpart()`, which mirrors R's own `model.frame.rpart.R` by re-invoking `rpart()` with the stored call's subset/method overridden.
`rpart_matrix.py`	An incorrect `patsy.dmatrix()`-based one-hot expansion of categorical predictors was removed and replaced with one numeric column per term, since rpart's C core natively handles multi-level categorical splits from a single integer-coded column—exactly as R's own `as.numeric(factor(x))` does, not `model.matrix`-style dummy coding.
Six files (`meanvar_rpart.py`, `printcp.py`, `snip_rpart.py`, `rsq_rpart.py`, `text_rpart.py`, `path_rpart.py`)	Each checked a fitted tree's class identity via the literal Python dunder `'__class__'` as a dict key (always `None`); corrected to the actual sentinel key, `'_rpart_class'`, that `rpart.py` sets.

Continued on next page

File(s)	Defect and Fix
Eight files (`predict_rpart.py`, `residuals_rpart.py`, `roc_rpart.py`, `summary_rpart.py`, `print_rpart.py`, `text_rpart.py`, `labels_rpart.py`, `xpred_rpart.py`)	All looked up `'ylevels'`/`'xlevels'` while `rpart.py` stores these under the R-attribute-style keys `'_ylevels'`/`'_xlevels'`; corrected uniformly, restoring classification-tree reporting and prediction across the board.
`rpartcallback.py`	An unbounded `_eval_result_keeper` list leaked one heap-allocated `SEXPREC` per node/split evaluation for the lifetime of every `method=4` fit; converted to a single-slot holder that frees the previous entry before storing a new one, with a `weakref.finalize`-driven session sentinel releasing the long-lived callback buffers once garbage-collected.
`importance.py`	An off-by-one in the surrogate-split row index misattributed surrogate-split importance to the wrong predictor whenever `maxsurrogate>0` (the default); corrected to match R's 1-based `seq_len(nsurr[i])`.
`roc_rpart.py`	Referenced `frame$splits`/`frame$yprob` columns that do not exist on `frame` in either language (confirmed empirically that the literal upstream R function silently operates on zero rows for every tree); rewritten to derive `truth`/`leaf_probs` directly from the `yval2` block that `rpart()` actually produces.
`predict_rpart.py`, `print_rpart.py`, `text_rpart.py`	`frame['yval2'].values` always returns a 1-D object array of per-row arrays—a pandas `Series.ndim` is always 1 regardless of contents—so every `type='matrix'`/`'prob'` prediction and every non-scalar print path silently failed; replaced with `np.array(frame['yval2'].tolist(), dtype=float)` at every call site.
DataFrame-vs-ndarray mismatches	`rsq_rpart.py`, `plotcp.py`, `prune_rpart.py`, `labels_rpart.py`, and `snip_rpart.py` all performed NumPy-style positional indexing on `cptable`/`splits`, which `rpart.py` constructs as `pandas.DataFrame` objects; corrected to `.to_numpy()`/`.iloc[]`, with `prune_rpart.py`'s pruned-`cptable` reconstruction additionally preserving original row labels via `.iloc[keep,:]`.
Missing cross-module imports	Ten modules called sibling-module functions without importing them (R's package namespace made them available automatically), producing `NameError` on first use; all imports added and verified acyclic.

Continued on next page

File(s)	Defect and Fix
zzz.py, rpartco.py, rpart_branch.py, plot_rpart.py, snip_rpart_mouse.py	rpart_env, R's package-level environment for passing per-device plotting parameters between the four plotting-family functions, was declared via global rpart_env in four modules with no backing definition anywhere; defined once as a shared dict in zzz.py and imported by the other four.
snip_rpart.py	A pandas Copy-on-Write chained-assignment bug (ff['ncompete'].iloc[newleaf] = 0) silently mutated a throwaway copy, never updating ff, corrupting importance()'s post-prune accounting; fixed via single-step .iloc[newleaf, ff.columns.get_loc('ncompete')] = 0.
summary_rpart.py	The R-mirroring parameter name object shadows Python's builtin object type; three internal np.array(..., dtype=object) calls were silently passing the fitted-tree dict as a NumPy dtype specification; fixed via import builtins and dtype=builtins.object.

Live execution during verification surfaced roughly sixteen further issues beyond the six audited-and-approved categories above—missing imports, DataFrame/ndarray type mismatches, pandas Copy-on-Write semantics, and R-value-semantics violations—none of which had been flagged by the static six-agent read, underscoring that dynamic verification (actually running the code) catches a materially different class of defect than static comparison alone. In total the pass touched 37 files (36 Python, 1 C) and fixed 28 distinct bug classes spanning 4 critical, roughly 8 high-severity, and roughly 16 issues found only through live execution.

11.4 The User-Defined Split Callback Bridge Repair

With the general audit complete, the user requested that the one remaining known limitation—method=4 failing inside init_rpcallback_c with RuntimeError: variable '(null)' not found—be root-caused and fixed. Tracing the literal error string led to compat_getVar's error() call inside src/rpart_callback.c, a file explicitly off-limits for modification. The actual defect, however, was one layer up: c_entry_points/init_rpcallback_c.c re-wrapped the opaque identity tokens rho_ptr, expr1_ptr, and expr2_ptr—the very pointers Python registers as dictionary keys in rpartcallback.py *before* calling into C—via make_opaque_sexp(), a helper that allocates a brand-new SEXPREC whose data field merely points at the original value. Because init_rpcallback() in the unmodifiable C source stores and later passes around the *new wrapper's own address*, every subsequent findVarInFrame/eval callback received an address that matched nothing in Python's lookup tables, regardless of how correctly the token had been registered. Since SEXP is simply a typedef for struct SEXPREC * and neither rho nor expr1/expr2 are ever dereferenced as real structs on this call path—only passed through opaquely to Python—a direct pointer cast is semantically sufficient, and was in fact the documented original intent of the file's own parameter-map comment. The fix removed the three make_opaque_sexp() calls in favor of

direct `(SEXP)` casts and removed their now-incorrect `free()` calls from the cleanup macro, since these are caller-owned pointers, not allocations made by this function.

Reproducing the original failure confirmed it resolved, but testing with a synthetic three-function user method (custom `init`/`split`/`eval` implementing an ANOVA-equivalent rule) immediately surfaced a segmentation fault inside `usersplit()`. The proximate trigger was a malformed, too-short `direction` array returned by the deliberately-broken test callback, but the underlying defect was in `rpartcallback.py` itself: `_py_eval()` caught all exceptions, including its own length-validation `ValueError`s, and returned a raw null SEXP that the unmodifiable C code dereferences unconditionally, crashing the interpreter instead of raising a catchable error. The fix hardens `eval1()`/`eval2()` to pre-compute the C-side-expected return length before invoking user code, so that any exception can be answered with a correctly-sized all-zero fallback array rather than an ill-formed one, while recording the exception in a `rho["_pending_exception"]` slot that `rpart.py` checks and re-raises immediately after the C call returns—turning a process crash into a normal Python traceback.

A third bug surfaced only when a realistic multi-step workflow (fit, then `printcp`, then `summary_rpart`, then `prune`, then `xpred_rpart`, all on the same object in one process) was run end to end: a `double free or corruption` abort, isolated by bisection to calling `xpred_rpart` after `prune`. The root cause was that `prune_rpart()`/`snip_rpart()` mutated the input tree dictionary in place and returned the same object, violating R's copy-on-modify value semantics; the original fit's cached `x`/`y` arrays, sized for the full dataset, became inconsistent with the now-pruned `frame`/`splits`/`csplit`, and `xpred_c`'s buffer-size assumptions—computed from that inconsistent state—corrupted the heap. The fix made `snip_rpart.py` copy the top-level dict and its `frame`, `splits`, `csplit`, and `where` entries independently at function entry, before any mutation, so neither `snip_rpart` nor, transitively, `prune_rpart` ever mutates a caller-visible object again.

After all three fixes, the full `pytest` suite (12/12) and an extensive manual regression script covering classification, regression, categorical predictors, pruning, prediction, plotting, cross-validation, and user-defined splits all passed, and `git status` confirmed that `r2py_rpart/src/` had never been touched—the sole C-level change was confined to `c_entry_points/init_rpcallback_c.c`, a directory the user had explicitly sanctioned for modification.

12 Phase 11: R Test-Suite Conversion and a Latent Control-Parameter Bug

12.1 Motivation

`rpart`'s own CRAN distribution ships fourteen R test scripts under `rpart/tests/`, covering formula edge cases, cost-weighted classification, case-weight scaling, user-defined splits, and cross-validated prediction—none of which had yet been ported to the Python side. Beyond simply growing test coverage, converting a package's own historical regression tests is a distinctively powerful correctness check: these scripts were written by the original package

authors specifically to pin down numerically exact behavior (`all.equal()` assertions to full floating-point precision) rather than to merely smoke-test the happy path, so translating them was expected to–and did–surface defects invisible to the more casual testing performed in Phases 9 and 10.

Validating a ported function against its original implementation is, in general, a substantially easier problem than testing new software from a blank specification: the source package itself is a complete, executable specification of the intended behavior, so a test-conversion agent can concentrate almost entirely on constructing representative and edge-case input parameters, with comparatively little need to hand-derive expected outputs, since the R implementation supplies them directly. The specific mechanism chosen to supply those reference outputs, however, is a deliberate design choice made differently here than in some sibling R-to-Python conversion projects: rather than calling a live R session at test-execution time (e.g., via `rpy2`), the reference values and fixture datasets used throughout this phase are captured once, from a live R session, into checked-in CSV files (Section 12), so that the resulting Python test suite carries no runtime dependency on an R installation being present. This trades the ability to silently pick up changes in a live-updated R reference for a self-contained, reproducible, dependency-free test suite appropriate for a package intended for standalone distribution.

12.2 Methodology

The `/convert-r-tests-to-python` (see Appendix M.1) skill was invoked:

```
/convert-r-tests-to-python rpart/tests/ rpart/ r2py_rpart/
```

in the `r-to-python` conda environment. The skill discovered 14 R files (`backticks.R`, `cost.R`, `cptest.R`, `minus_in_formula.R`, `priors.R`, `rescale.R`, `testall.R`, `treble.R`, `treble2.R`, `treble3.R`, `treble4.R`, `usersplits.R`, `xpred1.R`, `xpred2.R`) and dispatched the `@convert-r-test-to-python` agent once per file, deliberately sequentially rather than in parallel, so that each conversion could build on the fixtures and bug fixes established by the ones before it and so that shared library-state mutations during test execution would not race across concurrent agent processes. Each prompt supplied the R test path, the R library path, the Python package root, the output directory, the running list of already-created fixtures, and the running list of library fixes already applied, to prevent duplicated fixtures or accidental regressions. Midway through the batch (the `priors.R` conversion), an agent discovered that the `r-to-python` conda environment held a stale, non-editable installation of `r2py_rpart` that predated even earlier fixes made in this same session; every subsequent invocation was instructed to reinstall via `pip install -no-build-isolation .` as a standard first step before trusting any test result.

12.3 Per-File Conversion Highlights

Six of the fourteen conversions surfaced genuine, previously undetected library bugs; the remainder confirmed existing behavior correct. Table 4 summarizes each file.

Table 4: Per-file outcomes of the Phase 11 R test-suite conversion.

R File	Dataset	Outcome
`backticks.R`	synthetic iris	Confirmed `model_frame_rpart.py`'s existing backtick-stripping regex already mirrors R's; no library change needed.
`cost.R`	`survival::lung`	**Bug:** `rpart.py`'s `ymat` construction used `Y.T.ravel()` where R's `as.double(t(init$y))` is algebraically a plain row-major `Y.ravel()`; the erroneous transpose silently scrambled multi-column (`ny>1`) response pairing, invisible to every prior single-column-response test.
`cptest.R`	`mystate`	Verified `cp` scales correctly with dataset size across a 20× replicated variant; no bug.
`minus_in_formula.R`	`mtcars`	**Bug:** the formula parser only supported bare `"y ~ ."` or plain term lists; `"."` combined with `"-"` exclusions raised `KeyError`. Fixed with a top-level tokenizer implementing R's formula algebra.
`priors.R`	synthetic, 15 rows	**Bug:** `init["parms"]` was flattened with a plain row-major `.ravel()`; R's `unlist()` on a list containing a matrix uses the matrix's native column-major order, so a non-symmetric `loss` matrix was silently transposed before reaching `giniinit()`, corrupting the root node's predicted class.
`rescale.R`	synthetic, 9 obs.	Verified `rpart_exp`'s cumulative-hazard rescaling against a hand-computed table; no bug.
`testall.R`	`stagec`, `mystate`, `cu.summary`	**Seven bugs**, spanning `xval` scalar-vs-array handling, undersized C output buffers, a response cached before NA-filtering, categorical `newdata` re-encoding in prediction, per-element rounding, an NA-count reporting bug in three files, and `csplit` row truncation for multi-level categorical predictors.
`treble.R`	`stagec`	Verified weight-scaling of `frame$wt`/`dev`/`yval2`/`improve` on the hand-built `Surv` model-frame path; no bug.
`treble2.R`	`mystate`	Verified the formula+`data=`+`weights=` path's weight scaling, including a careful distinction between R's `rep(x,n)` tile semantics and `rep(x,length.out=n)` resize semantics; no bug.

Continued on next page

R File	Dataset	Outcome
`treble3.R`	`cu.summary`	Verified 10-class `yval2` column layout under weight scaling; no bug.
`treble4.R`	`kyphosis`	Verified a non-symmetric `loss` matrix's column-major fill under weight scaling; no bug.
`usersplits.R`	`mystate`	**Bug:** `xpred_rpart.py`'s parms-flattening handled only `dict`/`None`, raising `TypeError` whenever a user-defined `init()` returns a bare scalar `parms`, matching the built-in anova method's own convention.
`xpred1.R`	synthetic, 12 rows	Verified `return_all=True` on the anova (`numresp==1`) path; no bug.
`xpred2.R`	`stagec` (poisson)	**Bug:** the same row/column-major flattening defect as `cost.R`, recurring independently in `xpred_rpart.py`'s separate cross-validation code path, silently producing degenerate, root-node-constant output for any multi-column-response method.

After all fourteen agent invocations (`testall.R` alone contributed six pytest functions), the Python test suite had grown from 12 to 38 passing tests, with six new CSV fixtures (`lung.csv`, `mystate.csv`, `mtcars.csv`, `stagec.csv`, `cu_summary.csv`, `kyphosis.csv`) exported once from live R via `write.csv` to remove any runtime R dependency from the Python test suite. One item was left explicitly documented but unresolved at session end: a small, reproducible divergence in the trailing rows of cross-validated `xerror`/`xstd` in two of `testall.R`'s six fits, encoded in the test file as a loose sanity bound rather than an exact assertion, and investigated at length without a conclusive root cause—since the underlying `rpart.c`/`xval.c` C sources were confirmed byte-identical between the two packages.

12.4 Forensic Root-Cause Investigation of the Cross-Validation Discrepancy

The unresolved discrepancy documented above was pursued as a same-day follow-up, and its resolution is documented here in full because the investigative path—and the eventual, deliberately counter-intuitive conclusion—is itself instructive about the risks of attributing numerical divergence to floating-point non-determinism before ruling out simpler explanations.

12.4.1 Hypotheses Eliminated

Both a live R session and a live Python session were used side by side to reproduce the exact divergent cptable rows at ten-digit precision, confirming that `fit1` (poisson, `usesurrogate=0`, `cp=0`) matched R exactly through six of nine rows before diverging, and that `fit2`, built with the shallower default `cp=0.01` control, matched R exactly on every row including the

deepest—an important early clue that the defect was specific to very deep, lightly-pruned trees rather than a general property of the cross-validation code. A sequence of increasingly specific hypotheses were tested and eliminated with direct evidence rather than argument: (1) precision loss in the checked-in CSV fixture, ruled out by an exact byte-for-byte comparison against R's live in-memory dataset; (2) divergence between the two packages' C sources, ruled out by `diff -q` on `xval.c`, `rpart.c`, and `mysort.c`; (3) FMA (fused multiply-add) instruction contraction from differing compiler flags, ruled out by disassembling a minimal reproduction under all four flag combinations and finding no fused instructions were emitted in any case, since the toolchain's baseline instruction set predates FMA3 entirely; and (4), decisively, compiler optimization level and target-architecture flags in general, ruled out by temporarily rebuilding the actual Python extension with R's exact real compiler flags and observing the discrepancy was completely unchanged. A parallel check confirmed the deterministic, non-cross-validated full tree for `fit1` matched R node-for-node to the deepest leaf, confining the defect entirely to the cross-validation refit loop.

12.4.2 Bit-Level Instrumentation

With source-level and compiler-level causes eliminated, the investigation moved to binary-level instrumentation. Identical debug print statements were added to both a scratch, debug-instrumented local rebuild of R's own `rpart` package and to `r2py_rpart`'s copy of the same file, first in `xval.c` (dumping every fold's per-observation prediction-error array) and then, after finding that only 3 of 10 cross-validation folds diverged and only from a specific cp-column onward, in `bsplit.c` (dumping every candidate split's `improve`/`split` value in exact hex-float form) and finally in `partition.c` (dumping the boolean outcome, and every input to, the per-node "can I stop splitting now?" test). This last step was decisive: every logged field matched bit-for-bit between the two languages at every node, except one—every single line in the R log read `min_split=20`, and every single line in the Python log read `min_split=21`, a constant, node-independent, fold-independent integer difference of exactly one, present from the very first instrumented line. At the specific node where R's `20 < 20` test evaluates false (continue splitting) while Python's `20 < 21` test evaluates true (stop), the two languages' trees first diverge in shape—which fully explains every previously observed symptom in numerical terms, since divergence downstream of a differently-shaped tree in a numerically ill-conditioned cross-validation fold looks, superficially, exactly like floating-point noise.

12.4.3 Root Cause: A Sentinel-Clobbering Bug in the Control-Parameter Defaults

R's `rpart.control()` computes `minsplit`/`minbucket` defaults with `missing()`, which inspects the original call site: when neither argument is supplied, R leaves `minsplit` at its literal default of 20. `r2py_rpart`'s pre-fix emulation of `missing()` used an `_MISSING` sentinel object, but its three-branch control flow had a subtle defect: the first branch, computing `minbucket`'s default, reassigned the `minbucket` local variable to its newly-computed value before the second branch checked whether `minbucket` had originally been supplied—so the second branch always saw a non-sentinel value and incorrectly executed the back-derivation `minsplit =`

`minbucket * 3`, yielding `minsplit=21` instead of R's `20` whenever a caller supplied neither argument, the library's own documented default. The fix captures whether `minbucket` was originally missing into a separate boolean before it is reassigned, and checks that captured boolean afterward instead of re-inspecting the now-mutated variable—a minimal, three-line diff verified against R across all four meaningfully distinct argument combinations. This single defect fully explains why the discrepancy only ever appeared in maximally deep, `cp=0`-adjacent trees (only such trees ever reach a node with exactly 20 observations, the boundary the off-by-one affects), why only specific cross-validation folds were affected (whether any node in a given fold's held-out subset happens to land at exactly 20 rather than 19 or 24 observations is data-dependent), and why a separate, previously logged `usesurrogate=2` tree-size discrepancy (R producing 10 splits against Python's 9 for an otherwise identical fit) disappeared once the same one-line fix was applied.

12.4.4 Resolution and Retraction of a Secondary Hypothesis

Before the true root cause was found, a distinct latent defect had also been identified and provisionally treated as a contributing cause: `rundown.c`'s `oops:` fallback branch, reached when a held-out cross-validation observation is missing its split variable and `usesurrogate<2` forbids a surrogate fallback, correctly back-fills the `xpred[]` array for all remaining cp columns but only ever writes a single, mis-indexed `xtemp[]` element after the loop that was meant to cover it—leaving the true remaining `xtemp[]` slots holding stale values from whichever earlier, unrelated observation's call last wrote to the same shared buffer. This is confirmed to be a genuine defect present verbatim in both R's and `r2py_rpart`'s byte-identical `rundown.c`, and was initially credited with explaining why two of the thirteen originally-divergent (observation, fold) pairs disagreed between R and Python. Once the `minsplit` sentinel bug was fixed, however, R and Python build identical trees and process every fold's held-out observations in identical order, so even though the `rundown.c` defect still fires under the same conditions as before, it now leaves identical stale garbage in both languages' buffers rather than divergent garbage, and therefore produces no visible R-versus-Python difference at all—a conclusion directly confirmed by `fit1`'s full cptable matching R exactly on every row after only the `rpart_control.py` fix, with no change whatsoever to `rundown.c`. The `rundown.c` defect is accordingly retracted as an explanation for any part of the Phase 11 discrepancy—though it remains a real, independently verifiable latent bug in rpart's own upstream C source, and is deliberately left unfixed in `r2py_rpart` for the reasons given in the following paragraph.

The user was presented with the finding and the choice to keep the fix or revert to the pristine pre-investigation state, and chose to keep it. `test_testall.py`'s two previously loose sanity-bound assertions were tightened to check every cptable row at high precision against the now-exact reference values, and the module's explanatory docstring was rewritten to narrate the actual mechanism in place of the earlier, accurate-at-the-time but now-superseded description of an unresolved discrepancy. The full suite was reconfirmed at 38/38 after both the fix and the test-file tightening. Fixing the `rundown.c` defect itself was and remains out of scope: doing so would mean deliberately diverging from R's own, still-buggy behavior, which conflicts with a port whose stated goal is behavioral parity with R, bugs included.

13 Phase 12: Public-Interface Test Generation and Systematic Bug Resolution

13.1 Phase 12.1: R-Benchmarked Test Generation for the Public Interface

13.1.1 Motivation

Phases 10 and 11 fixed every bug that a manual audit or a historical R test suite happened to expose. Neither approach guarantees exhaustive coverage of the package's public interface: a systematic, per-function, R-benchmarked test generation pass closes that gap by producing positive, negative, and edge-case tests for every publicly documented entry point, each one verified directly against a live R session via `rpy2` rather than against a hand-derived expectation. The three test categories are defined as follows:

- **Positive tests:** cases where the input parameters are valid and the function is expected to return a correct result, covering the documented parameter combinations and typical use cases relevant to the function's intended behavior.
- **Negative tests:** cases where the input parameters are invalid or the function is documented to raise an error, covering every explicit error condition present in the R source—invalid parameter values, mismatched input lengths, and similar guard conditions.
- **Boundary and edge-case tests:** cases that probe the function's behavior at the limits of valid input or with special values—empty vectors, single-row data frames, `NA`/`NaN` inputs, and degenerate tree structures—that are not exercised by typical positive or negative cases but may reveal numerical-stability or special-casing defects.

This three-way coverage strategy is designed to maximize confidence in both the numerical correctness of the ported computation and the robustness of its error handling; as `r2py_rpart` is a numerical statistics package, broader software-engineering concerns such as concurrency or long-running stateful behavior fall outside the scope of this test suite.

13.1.2 Scope Determination

The command was invoked against `r2py_rpart/r2py_rpart/__init__.py` and `rpart/`, but a preliminary scan revealed a scoping ambiguity worth resolving before generating any tests: `__init__.py` itself only literally `def`ines 12 underscore-prefixed, low-level cffi/C-ABI wrapper functions, none of which correspond by name to any documented R public interface—they are the raw bindings underlying `rpart()`, `predict.rpart()`, `xpred.rpart()`, and `rpart.exp()`, not the public interfaces themselves. The file's actual `__all__` (36 entries) and its roughly 35 `from .module import name` statements re-export the true public surface from dedicated sibling modules. An `AskUserQuestion` was raised offering three interpretations of scope—testing only the four C wrappers with an R documentation counterpart, testing all 12 literal definitions regardless of R-doc match, or testing the roughly 35 imported public functions matching `rpart/man/*.Rd`—and the user selected the third. Cross-referencing

the remaining candidates against `rpart/man/rpart-internal.Rd` (which explicitly marks `pred.rpart`, `rpart.matrix`, `rpart.class`, `rpart.anova`, `rpart.poisson`, and `rpartco` as internal) and a commented-out `rpartcallback` alias in `rpart.Rd` narrowed the set to 23 confirmed public functions—reconciling the looser tally of nineteen R-documented aliases used earlier in Phase 10 with this port's flat namespace, in which the generic/S3-method pairs `post`/`post_rpart`, `prune`/`prune_rpart`, and `meanvar`/`meanvar_rpart` are each counted as two separate public functions, and `na.rpart` (omitted from the earlier tally) is confirmed public via `rpart/man/na.rpart.Rd`, which carries no `internal` keyword.

13.1.3 Methodology

The `/generate-python-file-tests` (see Appendix N.1) skill was invoked:

```
/generate-python-file-tests r2py_rpart/r2py_rpart/__init__.py rpart/
```

The `@generate-python-function-tests` agent was then invoked once per function, sequentially and in the foreground, over the 23-function filtered list, each run extending a single shared helper module, `r2py_rpart/tests/_r_rpart_helpers.py`, with dataset loaders, R/Python type-marshaling helpers, per-function R-oracle derivation replicas, and `Axes`-data-extraction utilities for the plotting-family functions, rather than each duplicating this infrastructure independently. A `tests/conftest.py` was added during the `plotcp` run to force matplotlib's non-interactive `Agg` backend package-wide. For the two generic/S3-method pairs where a wrapper is a literal one-line forward (`post`/`post_rpart`, `prune`/`prune_rpart`, `meanvar`/`meanvar_rpart`), the second agent in each pair correctly recognized the near-total overlap with the first and scoped itself down to only the behavior unreachable through the thin wrapper, chiefly positional-argument dispatch order, rather than duplicating test content.

13.1.4 Key Findings

The 23 runs added 69 new test files, growing the repository's test suite from the 38 tests it held at the close of Phase 11 to 846 tests in total (804 passing outright, 42 intentionally failing, each a verified, documented R-vs-Python behavioral divergence rather than a flawed test). At least nine genuine library defects were identified and left unfixed for a dedicated follow-up pass, summarized in the table below.

Function	Defect Documented
`rpart`	`RuntimeError: internal output buffer 'dnode' was undersized` on 0-row data or an all-NaN predictor column, instead of R's degenerate 1-row-frame output.
`predict_rpart`	`type="matrix"` on single-row `newdata` silently succeeds where R errors (NumPy never drops the row dimension); `_naresid()` over-expands rows even when R's actual dispatched `na.action` method is a documented no-op.
`print_rpart`, `printcp`, `summary_rpart`, `rpart_class`	Systemic decimal-padding and digit-rounding gaps: R's column-wide `format(signif(x,digits))` common-decimal-place alignment has no Python analogue in the port.
`rpartco.py`'s `compress()`	A copy-paste defect (`left["left"]` instead of `right["left"]`) corrupting boundary-vector lengths for deep, asymmetric trees, propagating into `plot_rpart` and `post`.
`post_rpart`	Default plot title indexes `tree['terms']['variables'][1]` (first predictor) instead of `[0]` (the response variable).
`residuals_rpart`	Classification `pearson`/`deviance` residuals crash via the same `yval2`-as-1-D-Series defect already worked around elsewhere.
`text_rpart`	`fancy=True` mode selects split-edge labels with mismatched masks against R's `!is.na()`-based selection, producing the wrong *values*, not merely a shape error.
`xpred_rpart`	A memory-corruption/segfault risk in the C bridge reachable via a bare-scalar `cp=` argument, isolated into a subprocess-based regression test to protect the main pytest process.

One process-resilience incident occurred: the `rsq_rpart` agent hit a transient API overload error mid-run and was resumed via `SendMessage` to the same agent identifier, completing cleanly rather than requiring a full restart.

13.2 Phase 12.2: Resolution of All Documented Bugs

13.2.1 Motivation

Phase 12.1 deliberately deferred every fix to keep test generation and bug fixing as separate, independently reviewable steps. This phase closes that loop, driving the 846-test suite to a full pass by resolving the root cause of every failure rather than weakening any assertion.

13.2.2 Methodology

The `/fix-bugs-found-in-tests` (see Appendix O.1) skill was invoked:

```
/fix-bugs-found-in-tests r2py_rpart/tests/ r2py_rpart/ rpart/
```

The single highest-priority failure, `rpartco.py`'s `compress()` defect from Phase 12.1, was fixed directly by hand rather than delegated, since its root cause (`templ` built from the wrong

branch's `left` array, confirmed against R's own `rpartco.R:127`) was already fully understood. A read-only `general-purpose` research agent was then dispatched to read every remaining failing test alongside its R source counterpart and produce a root-cause diagnosis grouped by shared underlying defect, yielding ten groups (A–J). The most consequential, Group A, was a single missing capability—no Python equivalent of R's `format()`/`as.character()` fixed-versus-scientific-notation decision procedure, which compares the rendered character width of both candidate representations rather than Python's exponent-based rule—manifesting independently across five separate files. Six `@fix-bug-found-in-test` agents were then dispatched together against disjoint file sets to permit safe concurrent execution, each fixing one cluster: `plotcp.py` (lty-code mapping and a deliberate design-choice correction to a test rather than the library), `printcp.py` (digit formatting, corrected default `digits`, input validation, an unconditionally-printed header), `print_rpart.py`/`rpart_class.py` (vector-wide decimal alignment, discovered by disassembling R's own bytecode via `Rscript` to find that R's `dev`/`yval` formatting call omits an explicit `digits=` argument on its outer `format()`), `summary_rpart.py` (the same formatting primitive applied to cut-points, complexity values, and the variable-importance block), `residuals_rpart.py` (a general NULL-propagation helper reproducing R's `frame[[column]][object$where]` semantics), `predict_rpart.py` (correct S3-dispatch-aware `naresid()` behavior and a reproduced R singleton-matrix quirk), and `__init__.py` (a buffer-size floor guaranteeing space for at least one degenerate root-only node regardless of input size).

13.2.3 Key Findings

A reusable formatting primitive emerged organically during this pass: the first agent's `r_format_double()`—an R-`format()`-equivalent width-comparison routine for a single value—was independently extended by three subsequent agents into `printcp.py`'s `_r_format()`, `print_rpart.py`'s vector-and-common-decimal-place `r_format_vector()`, and `summary_rpart.py`'s `_r_format_signif()`/`_r_format_matrix_column()`, avoiding four independent reimplementations of the same algorithm. One process incident occurred: an agent's own git-hygiene check misidentified the already-applied, in-scope `rpartco.py` fix as stray noise from an unrelated prior session and reverted it via `git checkout -`; this was caught immediately from a system-provided diff snapshot and reapplied before any further work propagated, after which every subsequent agent prompt explicitly fenced off files outside its own assigned scope. Two tests were determined, after empirical verification against live R, to be flawed rather than the library, and were corrected in place instead of changing library behavior to match them.

The final state, verified across four repeated full-suite runs including one against a freshly rebuilt, non-editable install, is 844 passed, 2 `xfail`ed, 0 failed. The two permanent `xfail(strict=True)` tests document a single, structurally unfixable divergence: R's `print.rpart`/`summary.rpart` call-echo relies on `dput()` rendering the *unevaluated* call expression captured by `match.call()` (the literal source token a caller typed, such as a bare variable name or an unexpanded sub-call), whereas `r2py_rpart.rpart()`'s stored `Call` dictionary necessarily holds already-*evaluated* argument values with no record of the caller's original source text—a gap that would require call-site source capture inside `rpart.py` itself

(for instance via `inspect`/AST introspection) to close, and was judged out of scope for this pass. A subsequent gap audit, run at the user's request, additionally cataloged 94 further "soft" divergences that emit a warning but do not fail any test—cases where both languages correctly reject the same invalid input but describe the rejection in each language's own native error vocabulary—grouped into seven themes (missing required arguments, low-level Python exceptions versus R's validation-flavored messages, invalid enum strings, non-numeric arguments to numeric operations, out-of-range indexing, plotting-internal error asymmetries, and miscellaneous unrelated-root-cause pairs), none of which represent an accept/reject behavioral defect on either side. This count (91 `assert_python_and_r_errors_agree` message-mismatch warnings plus 3 explicit `KNOWN GAP` warnings, one per affected test) was independently reproduced by re-running the full suite and grepping its captured warnings.

14 Phase 13: Example-Script Translation and Rendering-Parity Fixes

14.1 Motivation

With the library itself verified correct to 844/846 passing tests, this phase turns to a different kind of validation: translating two complete, realistic `rpart` demonstration workflows from R to Python and comparing their rendered output side by side. Unit tests verify individual functions in isolation against numeric or structural assertions; end-to-end example scripts instead exercise the full user-facing pipeline—data loading, formula-based fitting, cross-validated pruning, and PDF plotting—in the sequence an actual user would follow, and are uniquely suited to catching visual/rendering defects that no numeric assertion would ever check.

14.2 Translating the Example Scripts

`r2py_rpart/examples/regression_tree.R` reads a fourteen-column dataset with missing values, converts eight columns to factors, fits an anova tree with `cp=.0005`, selects a pruning cp via both the minimum-cross-validated-error rule and the 1-SE rule, and repeats the fit/prune/plot cycle on a bootstrap resample. Translating it to `regression_tree.py` surfaced one bug at the example-script (not package) level: naively casting the eight predictor columns to `"category"` dtype after they had been elevated to `float64` by NaNs elsewhere in the frame produced float-valued category labels (`"1.0"`, `"2.0"`, ...), whereas R's `as.factor()` on a numeric column stringifies via `as.character()` and never produces a trailing `.0`; this broke `labels_rpart.py`'s string-join logic downstream and was fixed by mapping each non-null value through `str(int(v))` before categorization. A newly authored `classification_tree.R`, fitting a classification tree on a renamed variant of Fisher's iris data stored in an R-only `iris_jun.RData` binary file, was translated identically; since no R-to-pandas binary reader was available in the target environment, the dataset was exported once via `write.csv` to a checked-in `iris_jun.csv` companion file, mirroring the role `income.data` already played for the regression example. At the user's request, both language pairs were standardized on a common output convention: every artifact filename prefixed with the script's

base name and suffixed `_R`/`_Python`, and all console output redirected to a matching `*_output_{R,Python}.txt` file (via R's `sink()` and Python's `contextlib.redirect_stdout` respectively).

14.3 Visual-Parity Bug Discovery and Fixes

The user independently regenerated both languages' PDFs and, by visual comparison, reported four rendering defects: overlapping, illegible x-axis tick labels on the cp-table plot; solid point markers where R renders hollow circles; split-condition text rendered to the right of each branch point rather than centered above it; and a bold title overlapping the tree drawing rather than sitting cleanly above it. With no PDF-to-image tool available except Ghostscript, both languages' PDFs were rasterized to PNG for direct visual inspection throughout this investigation. Root-causing each issue required going beyond the R source text to R's actual graphics-engine behavior: printing `rpart:::plotcp`'s live R source confirmed it contains no label-thinning logic at all, and a minimal, unrelated reproduction (plotting 30 points with 30 requested axis labels) confirmed the skipping of overlapping labels is an emergent behavior of R's low-level axis renderer, not something any `rpart` function implements; similarly, printing `rpart:::plot.rpart`'s live source confirmed that R's automatic axis padding is applied by the underlying `plot()` call regardless of `plot.rpart`'s own arguments, and is unrelated to the title-overlap symptom, which instead traced to `plot_rpart.py` having no title parameter at all and callers instead relying on matplotlib's much smaller default title padding.

Four fixes were applied at the package level. `plotcp.py` gained `_thin_overlapping_ticklabels()`, a helper that measures each tick label's rendered pixel extent via the matplotlib renderer and suppresses any label that would overlap the previous kept one; an initial implementation using `set_visible(False)` was replaced with `set_alpha(0.0)` after discovering, by reading matplotlib's own source, that `Axes.get_xticklabels()` silently drops—rather than merely renders invisibly—any label with `get_visible()` false, which would have broken this project's own test helpers that read back the full label set. The same file's point markers were switched to `markerfacecolor='none'`, reproducing R's hollow `pch=1` circle. `text_rpart.py`'s default label-drawing closure was given `ha='center'`, matching R's centered default text alignment; this was confirmed to carry zero regression risk since every existing test in that file supplies its own custom label-capturing closure specifically to bypass the default one. `plot_rpart.py` gained a new, previously absent `main` parameter that, when supplied, sets a bold axes title with generous padding, resolving the overlap without touching any axis-limit computation.

A candidate fix for the title-overlap issue—replicating R's roughly 4%-per-side automatic axis padding directly inside `plot_rpart.py`'s `set_xlim`/`set_ylim` calls—was considered and deliberately rejected, because doing so would change the actual values returned by `ax.get_xlim()`/`ax.get_ylim()`, against which several pre-existing Phase 12.1 tests assert exact equality to a tight-bounding-box arithmetic replica that intentionally omits real device-rendering padding. Grepping the existing test files for any reference to a title confirmed that adding a wholly new, previously nonexistent parameter was a strictly additive, zero-regression-risk fix, whereas altering axis limits was not—illustrating a recurring tension in this project between visual fidelity to R's rendered output and numerical fidelity to R's

documented, testable return values, resolved here in favor of the fix that could not regress the latter.

14.4 Verification

The three directly affected test files were run first (106 passed, 0 failed after the `set_alpha` correction), followed by the full repository suite (844 passed, 2 `xfail`ed, 0 failed, matching the Phase 12.2 baseline exactly). A user question, "Did you reinstall the package?", surfaced a workflow gap: interim verification had been performed against a copy of `r2py_rpart` shadowed in from the repository source directory by an accident of the shell's working directory, rather than the actually-installed site-packages distribution that `pip_install.sh` produces. A clean `pip install -no-build-isolation .` reinstall was performed and re-verified from a shadowing-immune `/tmp` working directory via direct introspection of the installed copy's source, and the full suite was re-run once more to confirm the identical 844/2/0 result. All four affected example PDFs were regenerated, rasterized, and visually re-confirmed to match their R-generated reference renders.

15 Discussion

The work documented in this report demonstrates that the seven-phase, skill-and-sub-agent AI-assisted conversion methodology validated on `KernSmooth`—a package whose computational core is reached exclusively through R's simple, array-based `.Fortran()`/`.C()` interface—extends successfully to a substantially harder case: a package whose entire computational core is reached through R's `.Call()` interface, which passes and returns opaque, R-specific `SEXP` objects rather than plain C arrays. The companion methodological framework explicitly identified `.Call()`-based packages as an open problem, noting that such a package would require either reimplementing the C-level computation from scratch (forfeiting the correctness guarantee of reusing validated source) or writing a thin SEXP-compatible C shim that bridges the `.Call()` interface to a Python-callable equivalent, and named `rpart` itself as the representative example of this harder class. Phases 1 through 5 of this project constitute exactly that shim: a from-scratch, standalone reimplementation of the relevant slice of R's C API (51 external items, 53 header files) sufficient to compile `rpart`'s original, unmodified computational C source outside of `libR.so`, plus five pure-C entry-point wrappers translating between the SEXP-based internal interface and a plain-array interface callable from Python via `cffi`. That this fake-header strategy succeeded—with the resulting 37 translation units (31 package sources, those five wrappers, and the Category E interpreter-item helper file added in Phase 9.3) compiling without errors and the resulting shared library reproducing R's cross-validated tree structure exactly, once the control-parameter default bug documented in Phase 11 was fixed—suggests that the `.Call()` barrier, while substantially more laborious than the `.Fortran()` case, is not a fundamentally different problem: it is the same reuse-the-original-source philosophy applied one layer further down the R runtime, at the cost of reimplementing (rather than merely bridging) the portion of the R C API that the target package actually exercises.

Two-track parallelism as a general pattern. The division of this project into a C-focused

track (Phases 1–5) and an R-focused track (Phases 6–9) was possible only because the structural analysis in Phase 1 revealed that the R-to-C interface is narrow: exactly five `.Call()` invocations connect all 36 R source files to all 35 C source files. This observation—made explicit and load-bearing rather than assumed—let the two tracks proceed largely independently, with the R-layer conversion in Phase 9 depending on the C track only through five fixed, already-stabilized function signatures. Packages with a wider or less disciplined R-to-C interface would not admit as clean a separation, and the applicability of this two-track pattern to a new package should be assessed empirically, via the same kind of dependency-graph analysis performed in Phase 1, before committing to a parallel work plan.

The value of a package's own historical test suite. Phase 11's conversion of `rpart`'s fourteen CRAN-shipped regression tests surfaced six previously undetected library bugs, and the subsequent forensic investigation of a residual cross-validation discrepancy traced a numerically subtle divergence—initially indistinguishable from floating-point non-determinism—to a single off-by-one defect in a control-parameter default. Neither Phase 9's conversion process nor Phase 10's six-way manual equivalence audit surfaced this defect, because neither exercised trees deep and unpruned enough to reach the specific 20-observation boundary condition it affects. This is strong evidence that a package's own historical test suite—written by the package's own authors specifically to pin down exact numerical behavior rather than to smoke-test the common case—is a distinctively high-value, and non-substitutable, source of test coverage for any AI-assisted porting effort, and should be prioritized for conversion whenever it exists, independent of how much other testing has already been performed.

Static audits versus dynamic verification. Phase 10's six-way parallel static audit, in which independent agents read R and Python source side by side without executing either, found real defects but also missed roughly sixteen further issues that surfaced only once realistic, multi-step workflows were actually executed—DataFrame/ndarray type mismatches, pandas copy-on-write semantics, and R-value-copy-semantics violations chief among them. This mirrors a general lesson applicable beyond this project: static code comparison and dynamic execution are complementary, not substitutable, verification strategies, and an AI-assisted conversion pipeline that relies on only one of the two should expect to carry forward a materially larger residual defect rate than one that budgets for both.

Scope and limitations. Several aspects of this project's scope bound the immediate generalizability of its specific findings. First, `rpart` is a *recommended* R package with no dependencies on other CRAN packages beyond base R and its own compiled C source; a package with a deep tree of CRAN dependencies would require the same phased methodology to be applied recursively to each dependency, and the cross-project cost of doing so—in agent invocations, wall-clock time, and the size of the shared language-dependency guide catalog—has not been characterized here. Second, several genuinely R-specific behaviors—most notably R's `dput()`-based unevaluated-call-expression printing in `print.rpart`/`summary.rpart`, and R's column-wise `format()`/`signif()` common-decimal-place alignment—remain either permanently unfixable without call-site source capture (the two `xfail` tests carried forward from Phase 12.2) or were only reproduced through substantial reverse-engineering effort (the `r_format_double()` family of helpers described in Section 13.2). A future methodology could reduce this class of defect by cataloging R's introspection- and formatting-primitive semantics

as their own dedicated category of language-dependency guide, generated once and reused across projects, rather than rediscovering them ad hoc within each. Third, the interactive, mouse-driven `snip.rpart.mouse` function and the `eval.parent(model.frame(...))` dynamic model-frame construction pattern were both deliberately left as `NotImplementedError` stubs, since neither has a faithful Python equivalent without, respectively, a GUI event loop or a full R-style formula-evaluation environment; closing these gaps was judged out of scope for a port whose stated target is programmatic, non-interactive use.

Future work. The most direct extension of this project is closing the two `xfail` tests and the 94 cataloged soft divergences documented in Phase 12.2, most of which are differences in error-message vocabulary rather than accept/reject behavior. More broadly, the fake-R-C-API-header strategy developed in Phases 3–5 is itself a reusable artifact: the 51 fake header implementations cover a substantial fraction of the R C API surface that any `.Call()`-based package is likely to use (`SEXP` construction and accessors, `PROTECT`/`UNPROTECT`, memory allocation, error/warning signaling), and a natural next step is to extract this header set into a standalone, package-independent library that a future `.Call()`-based conversion project could adopt directly, needing to author only the incremental fake headers specific to R API surface not already covered here—in the same way that the 172 language-dependency conversion guides generated in Phase 8 are, in principle, directly reusable by any future project translating base-R idioms to Python.

Declarations

Data and code availability

The Python package `r2py_rpart` and all project artifacts (C and R dependency-graph analyzes, fake-header implementation guides, language-dependency conversion guides, per-function conversion results, and the test suite) are available at `https://github.com/r2py-project/python-rpart`. The Python package is additionally available as a standalone repository at `https://github.com/r2py-project/r2py_rpart`. The original R `rpart` package (version 4.1.27) is available from CRAN under a GPL-2/GPL-3 license.

Author contributions

Y.C. designed the conversion methodology, implemented all phases, wrote the code, and wrote the manuscript.

Competing interests

The author declares no competing interests.

Acknowledgements

The author thanks the HPC facility of the University of Notre Dame for providing computational resources. The AI-assisted development documented in this report was performed

using Claude Code (Anthropic).

A Phase 1: C Dependency Analysis

A.1 Skill: /analyze-c-folder-dependencies

```
---
name: analyze-c-folder-dependencies
description: Analyzes all C files in a folder to extract structural dependency
    information file by file, categorizing calls strictly into internal and external
    dependencies.
---

# Analyze a C Folder for Dependencies

## Description

When provided with a target folder, your task is to recursively scan for all C files (
    including the header files), use the `@analyze-c-file-dependencies` agent on each
    file individually, and save the results as structured JSON files.

## Execution Steps

### Step 1: Scan for C Files

Recursively scan the provided target folder and all of its subdirectories for files
    with `.c` or `.h` extensions. Compile a complete list of their absolute file paths.

### Step 2: Process and Analyze

Iterate through the list of identified C files sequentially. For each file:
1. Invoke the `@analyze-c-file-dependencies` agent against the file and record its
    final JSON output.
2. If the command fails or throws an error for a specific file, log the error to the
    console and immediately continue to the next file in your list. Do not halt the
    entire operation.

### Step 3: Save Output

For every successfully analyzed C file, save the resulting JSON output according to
    these strict rules:
* **Naming Convention:** Use the exact name of the original C file, but append the
    extension `.json` (e.g., `init.c` becomes `init.c.json`).
* **Output Location:** If the user did not specify the output folder, save the new `.
    json` file in `c_refactor_analysis/{folder_name}/`, where the `{folder_name}` is
    the name of the target folder. The relative path must be the same as its
    corresponding original C file.
```

A.2 Agent: @analyze-c-file-dependencies

```
---
```

```
name: analyze-c-file-dependencies
description: Analyzes a provided C file to extract structural dependency information
    function by function (including function-like macros), categorizing calls strictly
    into internal and external dependencies.
---

# Analyze a C File for Dependencies

## Description

When provided with a C file (‘.c‘ or ‘.h‘), your task is to parse the file, identify
    every user-defined function or function-like macro, and trace all function-like
    calls (functions and function-like macros) made within their bodies. You must
    categorize these dependencies into strict groups and output the result as a
    structured JSON object.

## Execution Steps

### Step 1: Extract Function or Macro Declarations

Scan the target C file and identify every user-defined function or function-like macro
    defined within it (e.g., ‘int my_func() { ... }‘). Create a list of these function
    or macro identifiers.

### Step 2: Analyze Function or Macro Bodies and Trace Calls

For every function or macro identified in Step 1, thoroughly analyze its execution body
    . Extract the identifier of every function or function-like macro invoked within
    that body.

**Exclusions:**
* Completely ignore functions or macros that are part of the C standard library (e.g.,
    ‘printf‘, ‘malloc‘, ‘strlen‘, ‘size_t‘).
* Extract **only the identifier** (e.g., ‘PyArg_ParseTuple‘, ‘.Call‘), never the
    arguments or the parentheses.

Categorize the extracted identifiers into the following two mutually exclusive lists.
    You may need to traverse and scan other files in the same project folder according
    to the included header files, and you have to use the following heuristics:

* **Internal Dependencies:**
    * *Definition:* Identifiers for functions or macros defined within the same C file,
     or those that appear in other ‘.c‘ or ‘.h‘ files within the same project folder,
    which are included via ‘#include‘ directives into the target file.
* **External Dependencies:**
    * *Definition:* Identifiers for functions or macros belonging to known external,
    third-party libraries or language bindings (for example, R API calls like ‘
    Rf_protect‘, Python C API calls like ‘Py_Initialize‘, etc.). Only if you cannot
    find the definition of the function or macro in the same file or any other file in
    the same project folder, you should classify it as an external dependency.

### Step 3: Generate JSON Output
```

```
Construct a JSON object where each key is the name of a function or macro identified in
     Step 1. The value for each key must be an object containing exactly two arrays of
    strings: ‘internal_dependencies‘ and ‘external_dependencies‘. Deduplicate all
    identifiers within these arrays.

## Output Format Schema

You must output valid JSON strictly adhering to this structure. Do not include markdown
     code blocks if the system expects raw JSON.

{
  "my_custom_algorithm": {
    "internal_dependencies": ["helper_function_within_file", "project_utils_init"],
    "external_dependencies": ["Rf_protect", "PyArg_ParseTuple"]
  },
  "helper_function_within_file": {
    "internal_dependencies": [],
    "external_dependencies": []
  }
}
```

B Phase 2: R External Item Extraction

B.1 Skill: /extract-c-folder-r-extern-items

```
---
name: extract-c-folder-r-extern-items
description: Recursively processes a folder of C files to generate detailed CSV reports
     of R external references (macros, types, functions).
---

# Extract R External Items from a C Folder

## Description

When provided with a target directory containing C source or header files, your task is
     to process every C file (‘.c‘ or ‘.h‘) recursively. You will invoke the ‘@extract-
    c-file-r-extern-items‘ agent on each identified C file and save the extracted
    dependency data strictly as ‘.csv‘ files in a mirrored output directory.

## Execution Steps

### Step 1: Scan for C Source and Header Files

Recursively scan the provided target directory and all of its subdirectories. Compile a
     complete list of absolute file paths for every file with a ‘.c‘ or ‘.h‘ extension.

### Step 2: Sequential Processing and Error Isolation

Iterate through the compiled list of identified C files sequentially. For each file:
1. **Execute File-Level Agent:** Invoke the ‘@extract-c-file-r-extern-items‘ agent,
    providing it with the current ‘.c‘ or ‘.h‘ file path as its target.
```

```
2. **Error Handling:** If the sub-agent fails, times out, or throws an error for a
    specific file, log a clear warning to the console containing the file's path.
    Immediately proceed to the next C file in the list. Do not halt the entire batch
    execution due to individual file failures.

### Step 3: Save Output as CSV

For every successfully analyzed C file, save the tabular string output returned by the
    sub-agent according to these strict rules:
* **Data Format:** The output content must be saved strictly as a flat CSV file
    matching the schema defined by the sub-agent.
* **Naming Convention:** Use the exact base name of the original C file, but append it
    with `.csv` (e.g., `rpart.c` becomes `rpart.c.csv` and `rpart.h` becomes `rpart.h.
    csv`).
* **Output Directory Mapping:** Maintain the original directory hierarchy. Save the `.
    csv` file into the user-specified output directory, mirroring the exact relative
    path of the source C file relative to the input target folder.
* **Default Output Directory:** If the user did not explicitly specify an output
    directory, create a default directory path named `r_extern_analysis/{
    target_folder_name}/` (where `{target_folder_name}` is the base name of the target
    folder being scanned) and mirror the relative paths there.
```

B.2 Agent: `@extract-c-file-r-extern-items`

```
---
name: extract-c-file-r-extern-items
description: Extracts detailed location and usage data for R external references (
    macros, types, functions) in a C file.
---

# Extract R External Items from a C File

## Description

When provided with a C source file (`.c` or `.h`), your task is to locate every
    instance where an R external item is referenced. An R external item is defined as
    any type identifier, struct, macro, or function name that is declared in a header
    file located in the folder `~/.conda/envs/r-to-python/lib/R/include/`.

You must extract the exact line numbers, the containing code statements, and the
    specific R header file where each item is declared. The final output must be a
    strictly formatted, sorted CSV.

## Execution Steps

### Step 1: Identify Non-Local C Identifiers

Parse the provided C file to identify all macros, type definitions, structs, and
    function calls. Exclude any identifiers that are clearly defined within the
    provided C file itself or are part of the standard C library (e.g., `stdio.h`, `
    stdlib.h` elements).

### Step 2: Cross-Reference with R Headers
```

````
For the identifiers flagged in Step 1, use your search tools (e.g., `grep` or file-
    reading capabilities) to search both the C files that the current C file includes
    and the directory `~/.conda/envs/r-to-python/lib/R/include/`. Identify which of
    these items are declared in those R header files in `~/.conda/envs/r-to-python/lib/
    R/include/`.

For every matched R external item, record the following six data points:
* **`external_item`**: The exact name of the referenced item (e.g., `SEXP`, `PROTECT`,
    `ALLOC`).
* **`header_file`**: The specific header file within `~/.conda/envs/r-to-python/lib/R/
    include/` where the item is declared (e.g., `R.h`, `Rinternals.h`).
* **`category`**: A string flag indicating the category of the item, which can only be
    among "type", "variable", "function". Types include those defined by typedefs,
    structs, enums, or type-like macros. Variables are identifiers that represent
    values. Functions are identifiers that are invoked with parentheses `()`, which can
     be either functions or function-like macros. Note that there is no category for
    macros since they are always crafted to mimic the behavior of the three categories
    above.
* **`file_name`**: The name of the provided `.c` or `.h` file you are currently
    analyzing.
* **`line_number`**: The exact line number in the analyzed file where the statement
    containing the reference begins.
* **`context_statement`**: The complete code snippet containing the reference.
    * If the statement spans multiple lines, capture the entire complete statement from
     start to the terminating semicolon `;`, or the complete function prototype/macro
    definition.
    * Consolidate multi-line statements into a single string by escaping internal line
    breaks or replacing them with spaces.

### Step 3: Format and Sort Output

Compile the extracted data points into a CSV format. Sort the output primarily by `
    line_number` (ascending), and secondarily by `external_item` (alphabetically).

## Output Format Schema

You must output valid CSV data strictly adhering to the structure below. Include the
    exact header row. **CRITICAL:** You must enclose the `context_statement` strings in
     double quotes (`"`) to prevent commas or formatting elements within the C code
    from breaking the CSV structure.

```csv
external_item,header_file,category,file_name,line_number,context_statement
SEXP,R.h,type,rpart.c,40,"SEXP x = PROTECT(ALLOC(REALSXP, n));"
PROTECT,R.h,function,rpart.c,40,"SEXP x = PROTECT(ALLOC(REALSXP, n));"
ALLOC,R.h,function,rpart.c,40,"SEXP x = PROTECT(ALLOC(REALSXP, n));"
REALSXP,Rinternals.h,type,rpart.c,40,"SEXP x = PROTECT(ALLOC(REALSXP, n));"
```
````

C Phase 3: Fake Header Implementation Guides

C.1 Skill: /generate-r-extern-fake-guides

````
---
name: generate-r-extern-fake-guides
description: Orchestrates the batch generation of fake C++ header implementation guides
     for R's C API external items, enabling R package C source files to compile and run
     without linking against libR.so.
---

# Generate R External Item Fake Header Guides

## Description

When provided with a target `base_folder`, a CSV file matching the schema below, and a
    target `output_directory`, your task is to orchestrate the batch processing of this
     dataset. You must segment the CSV by unique `external_item` values and iteratively
     invoke the `@generate-r-extern-fake-guide` agent for each subset to generate the
    resulting guides to the specified output directory.

The goal of each guide is to document how to implement a drop-in C++ fake for a given R
     C API external item -- a type, macro, constant, or function -- so that the
    original package C source files can be compiled and linked without `libR.so`, and
    called directly from Python.

### Expected CSV Schema

Example:
```csv
external_item,header_file,category,file_name,line_number,context_statement
DL_FUNC,R_ext/Rdynload.h,type,init.c,12,"static const R_CallMethodDef CallEntries[] = {
     {""init_rpcallback"", (DL_FUNC) &init_rpcallback, 5}, {""rpart"", (DL_FUNC) &rpart
    , 11}, {""xpred"", (DL_FUNC) &xpred, 15}, {""rpartexp2"", (DL_FUNC) &rpartexp2, 2},
     {""pred_rpart"", (DL_FUNC) &pred_rpart, 12}, {NULL, NULL, 0} };"
DllInfo,R_ext/Rdynload.h,type,init.c,23,R_init_rpart(DllInfo * dll)
SEXP,Rinternals.h,type,rpart.c,41,"SEXP rpart(SEXP ncat2, SEXP method2, SEXP opt2, SEXP
     parms2, SEXP xvals2, SEXP xgrp2, SEXP ymat2, SEXP xmat2, SEXP wt2, SEXP ny2, SEXP
    cost2)"
INTSXP,Rinternals.h,type,pred_rpart.c,139,"SEXP where = PROTECT(allocVector(INTSXP, n))
    ;"
REALSXP,Rinternals.h,type,rpart.c,241,"cptable3 = PROTECT(allocMatrix(REALSXP, xvals >
    1 ? 5 : 3, rp.num_unique_cp));"
PROTECT,Rinternals.h,macro,rpart.c,194,"which3 = PROTECT(allocVector(INTSXP, n));"
allocVector,Rinternals.h,function,rpart.c,194,"which3 = PROTECT(allocVector(INTSXP, n))
    ;"
error,R_ext/Error.h,function,rpart.c,203,"error(""%s"", errmsg);"
R_alloc,R_ext/Memory.h,function,rpart.c,123,"rp.xdata = (double **) ALLOC(rp.nvar,
    sizeof(double *));"
```

## Execution Steps

### Step 1: Identify Unique Dependencies
````

```
Read and parse the input CSV file. Extract a definitive list of all unique values
    present in the ‘external_item‘ column (e.g., ‘DL_FUNC‘, ‘SEXP‘, ‘INTSXP‘, ‘PROTECT
    ‘, ‘allocVector‘, ‘error‘, ‘R_alloc‘).

*Note: Assume the CSV is pre-sorted. Identical external items are grouped sequentially
    to allow for efficient linear processing.*

### Step 2: Extract Segments and Invoke Sub-agents

Iterate sequentially through the list of unique ‘external_item‘ values one after
    another. Sequential execution is mandatory here: foundational fake definitions (e.g
    ., the ‘SEXP‘ struct, the ‘SEXPTYPE‘ enum, the arena allocator) must be generated
    before the items that depend on them (e.g., ‘allocVector‘, ‘INTEGER‘, ‘REAL‘), and
    each newly generated guide may be referenced by the agent processing the next item.
     For each unique item, strictly execute the following sub-steps:

1. **Extract the CSV Subset:** Isolate all rows where the ‘external_item‘ column
    matches the current target. Prepend the exact CSV header row (‘external_item,
    header_file,category,file_name,line_number,context_statement‘) to this subset to
    create a valid, standalone CSV-formatted string.
2. **Execute Fake Guide Agent:** Invoke the ‘@generate-r-extern-fake-guide‘ agent. Pass
     the target ‘base_folder‘, the newly created CSV subset string, and the specified
    output directory as inputs.
3. **Strict Error Handling:** If the agent execution fails, times out, or throws an
    error at any point, you must:
    - Log a precise error message to the console: ‘ERROR: Failed to generate fake guide
     for {external_item}. Proceeding to next item.‘
    - Immediately continue the loop with the next unique ‘external_item‘ in the list.
    Do not halt or abort the overall batch execution.
```

C.2 Agent: `@generate-r-extern-fake-guide`

```
---
name: generate-r-extern-fake-guide
description: Generates a fake C++ header implementation guide for a specific R C API
    external item, enabling the original R package C source files to compile and run
    without linking against libR.so.
---

# Generate an R External Item Fake Header Guide

## Description

When provided with a target ‘base_folder‘ and a CSV text input, your task is to
    generate a comprehensive fake header implementation guide for the specified ‘
    external_item‘ (e.g., ‘SEXP‘, ‘INTSXP‘, ‘allocVector‘, ‘error‘, ‘R_alloc‘). This
    guide documents how to implement a drop-in C++ replacement for that item so that
    the original package C source files compile and link without ‘libR.so‘, and can be
    called directly from Python.

The input CSV will strictly adhere to the following example schema:
‘‘‘csv
```

````
external_item,header_file,category,file_name,line_number,context_statement
allocVector,Rinternals.h,function,rpart.c,194,"which3 = PROTECT(allocVector(INTSXP, n))
    ;"
allocVector,Rinternals.h,function,rpart.c,241,"cptable3 = PROTECT(allocMatrix(REALSXP,
    xvals > 1 ? 5 : 3, rp.num_unique_cp));"
allocVector,Rinternals.h,function,rpart.c,261,"dnode3 = PROTECT(allocMatrix(REALSXP,
    nodecount, (3 + rp.num_resp)));"
allocVector,Rinternals.h,function,rpartexp2.c,47,"SEXP keep = PROTECT(allocVector(
    INTSXP, n));"
```

The generated guide must:
1. Define the `external_item`'s core functionality within R's C API.
2. Analyze its specific usage within the provided CSV dataset, incorporating surrounding source code context.
3. Provide a complete C++ fake implementation strategy governed by the three invariants below, including concrete, syntax-highlighted C++ code for every distinct usage pattern found in the CSV.

### Three Invariants That All Fake Implementations Must Respect

These invariants apply to every guide generated by this agent, regardless of the specific `external_item`. They are non-negotiable design constraints:

**Invariant 1 -- C++ error and warning style.**
All R error and warning mechanisms (`Rf_error`, `Rf_warning`, and their `#define` aliases `error`, `warning`) must be replaced by pure C++ equivalents. Errors (`Rf_error`) must throw a C++ exception -- either `std::runtime_error` or a project-defined subclass `RError : public std::runtime_error`. Warnings (`Rf_warning`) must write to `stderr` via `std::fprintf(stderr, "Warning: ...")`. Under no circumstances may `longjmp`, `setjmp`, R condition handlers, or `abort()` be used.

**Invariant 2 -- Arena-based memory management without modifying original code.**
All R memory allocation functions used in the package C source (`R_alloc`, `R_chk_calloc`, `R_chk_realloc`, `R_chk_free`, `S_alloc`, and the macros `ALLOC`, `CALLOC`, `R_Calloc`, `R_Realloc`, `R_Free`, `Memcpy`) must be reimplemented using a thread-local arena allocator. The arena is structured as a stack of frames: each frame corresponds to one invocation of a top-level `.Call`-style C function. An `ArenaFrame` RAII guard is pushed at function entry and popped (freeing all frame allocations) at function exit. The original C source files are **not modified** -- the arena is completely transparent to them because the same function names and macro names are preserved in the fake headers.

The canonical arena interface that all memory-related guides must reference is:

```cpp
// fake_arena.hpp  (foundational; generated once and referenced by all memory guides)
#include <cstdlib>
#include <cstring>
#include <stdexcept>
#include <vector>

struct ArenaBlock {
    char   *data;
```

```
    size_t  used;
    size_t  capacity;
};

struct Arena {
    std::vector<ArenaBlock> blocks;
    static constexpr size_t kDefaultBlockSize = 65536;

    void *alloc(size_t bytes) {
        bytes = (bytes + 7) & ~size_t(7);  // 8-byte align
        if (blocks.empty() || blocks.back().used + bytes > blocks.back().capacity) {
            size_t cap = bytes > kDefaultBlockSize ? bytes : kDefaultBlockSize;
            ArenaBlock b;
            b.data     = static_cast<char *>(std::malloc(cap));
            if (!b.data) throw std::bad_alloc();
            b.used     = 0;
            b.capacity = cap;
            blocks.push_back(b);
        }
        char *ptr = blocks.back().data + blocks.back().used;
        blocks.back().used += bytes;
        return ptr;
    }

    void reset() {
        for (auto &b : blocks) std::free(b.data);
        blocks.clear();
    }

    ~Arena() { reset(); }
};

// One arena per thread; stack discipline maps to nested .Call invocations
inline thread_local std::vector<Arena> gArenaStack;

struct ArenaFrame {
    ArenaFrame()  { gArenaStack.emplace_back(); }
    ~ArenaFrame() { gArenaStack.back().reset(); gArenaStack.pop_back(); }
};

inline void *arena_alloc(size_t bytes) {
    return gArenaStack.back().alloc(bytes);
}
inline void *arena_calloc(size_t n, size_t size) {
    void *p = arena_alloc(n * size);
    std::memset(p, 0, n * size);
    return p;
}
```

**Invariant 3 -- Best-effort faking of R interpreter items.**
Some R C API items (‘eval‘, ‘findVar‘, ‘findVarInFrame‘, ‘install‘, ‘R_getVar‘, ‘R_GlobalEnv‘, ‘R_NaString‘, environments ‘ENVSXP‘, and expression objects ‘LANGSXP‘/‘EXPRSXP‘) require a running R interpreter and cannot be fully faked. For these

```
    items, the guide must:
- Clearly mark the item as an **R Interpreter Item** at the top of the guide.
- Explain exactly why a complete fake is impossible.
- Provide the best-effort Python interop alternative: a C function pointer that Python
    registers via `ctypes.CFUNCTYPE` before calling the enclosing `.Call` function. The
     fake implementation calls this function pointer instead of `eval(expr, rho)` or `
    findVar(sym, rho)`. The guide must show the full C++ stub (the function pointer
    declaration, the registration function, and the stub body) and the corresponding
    Python-side `ctypes` registration code.
- Note which specific package methods or code paths require the item (e.g., "method=4
    user-defined splits in rpart") and state that those paths remain unavailable unless
     the Python function pointer is registered.

## Execution Steps

### Step 1: Ingest Inputs and Gather Deterministic Context

- **Path Resolution:** Iterate through each row in the CSV. Construct the absolute file
     path by concatenating the provided `base_folder` with the `file_name`.
- **File Inspection:** Navigate to the exact `line_number` in each target file. Read at
     least 15 lines above and 15 lines below the target line. Use the chosen window to
    understand: the C types of all arguments and return values, the scope and lifetime
    of variables, how the target `external_item` interacts with adjacent R API calls (
    especially `PROTECT`/`UNPROTECT`, `ALLOC`, and SEXP accessors), and what data flows
     into and out of the item.
- **Header Resolution:** You must read the local header file at `~/.conda/envs/r-to-
    python/lib/R/include/{header_file}` whenever the `external_item`'s struct
    definition, macro expansion, or function signature is not explicitly clear from the
     source code window. Also check `~/.conda/envs/r-to-python/lib/R/include/R_ext/`
    for sub-headers included transitively. If a header is not found in either `
    base_folder` or `~/.conda/envs/r-to-python/lib/R/include/`, explicitly state that
    the header is missing and proceed with the information available. Never search
    other locations.

### Step 2: Resolve External Dependencies

If local headers do not provide complete definitions, consult the following references
    in order:
1. `.Call/.External` documentation: `https://search.r-project.org/R/refmans/base/html/
    CallExternal.html`
2. Writing R Extensions (memory and error chapters): `https://cran.r-project.org/doc/
    manuals/R-exts.html`
3. Package-specific CRAN documentation (e.g., `https://cran.r-project.org/web/packages/
    rpart/`).

*Requirement:* Before writing the fake implementation, search the output directory for
    any previously generated fake guides. If a guide already exists for a type or
    function that the current `external_item` depends on (e.g., `allocVector` depends
    on `SEXP` and `SEXPTYPE`; `INTEGER` depends on `SEXP`; `error` depends on `RError`)
    , read that guide and ensure the new fake implementation is consistent with -- and
    references -- the already-established fake definitions.

### Step 3: Design the Fake C++ Implementation
```

```
Classify the `external_item` into exactly one of the five implementation categories
    below. Apply the rules for that category to design the fake.

**Category A -- Type or Enum Constant** (e.g., `SEXP`, `SEXPTYPE`, `INTSXP`, `DL_FUNC`,
     `DllInfo`, `Rboolean`, `R_len_t`, `Rbyte`):
- Provide a complete C++ struct, typedef, or `enum` definition that matches the
    observable behavior of the R type in the package source.
- For `SEXP` specifically: define `SEXPREC` as a C++ struct with at minimum a `SEXPTYPE
     type`, `int length`, `int nrow`, `int ncol`, and a `void *data` field. `SEXP` is `
    SEXPREC *`. SEXP objects are heap-allocated and not garbage-collected in the fake
    runtime; the caller owns their lifetime.
- For `DllInfo` and `R_CallMethodDef`: define as minimal structs or no-op placeholders
    sufficient for `init.c` to compile; the registration functions are no-ops.

**Category B -- Accessor Macro or Inline Function** (e.g., `INTEGER`, `REAL`, `LOGICAL
    `, `RAW`, `COMPLEX`, `asInteger`, `asReal`, `asLogical`, `LENGTH`, `XLENGTH`, `
    nrows`, `ncols`, `TYPEOF`, `STRING_ELT`, `VECTOR_ELT`, `CHAR`, `PRINTNAME`, `R_CHAR
    `):
- Implement as a C++ `inline` function (preferred) or as a `#define` macro if the
    original is a macro with no safe inline equivalent.
- The implementation casts `sexp->data` to the appropriate pointer type and returns it
    (for vector accessors) or extracts a scalar (for `asInteger`, `asReal`).
- For `nrows` and `ncols`: return `sexp->nrow` and `sexp->ncol`.

**Category C -- Allocation or Memory Function** (e.g., `allocVector`, `allocMatrix`, `
    R_alloc`, `R_chk_calloc`, `R_chk_realloc`, `R_chk_free`, `R_Calloc`, `R_Free`, `
    Memcpy`, `mkChar`, `PROTECT`, `UNPROTECT`):
- All heap memory obtained by SEXP-allocating functions (`allocVector`, `allocMatrix`,
    `mkChar`) must use `std::malloc`/`new` and must set `sexp->type`, `sexp->length`, `
    sexp->nrow`, `sexp->ncol`, and `sexp->data` correctly.
- All arena-allocated functions (`R_alloc`, `R_chk_calloc`, `R_chk_realloc`, `S_alloc`)
     must delegate to `arena_alloc` / `arena_calloc` from `fake_arena.hpp` (Invariant
    2).
- `R_chk_free` and `R_Free` free heap memory (for non-arena allocations). They must NOT
     free arena memory; the arena does that at frame destruction.
- `PROTECT` and `UNPROTECT` are **no-ops** in the fake runtime (there is no garbage
    collector). Implement as empty inline functions or identity macros. `
    PROTECT_WITH_INDEX` and `REPROTECT` are also no-ops.
- For every allocation function, explicitly show how `ArenaFrame` must be declared at
    the entry of each top-level `.Call` function to ensure cleanup (referencing
    Invariant 2).

**Category D -- Error, Warning, or Print Function** (e.g., `Rf_error`, `Rf_warning`, `
    Rprintf`, `REprintf`):
- Apply Invariant 1 strictly.
- `Rf_error(fmt, ...)`: format the message with `vsnprintf` into a `std::string`, then
    `throw RError(msg)` where `RError` is defined as `struct RError : public std::
    runtime_error { using std::runtime_error::runtime_error; };`. The `#define error
    Rf_error` alias from `R_ext/Error.h` still works unchanged.
- `Rf_warning(fmt, ...)`: format and write to `stderr` via `std::fprintf`. Do not throw
    .
- `Rprintf(fmt, ...)`: delegate to `std::printf`.
- `REprintf(fmt, ...)`: delegate to `std::fprintf(stderr, ...)`.
```

```
- Show how the `.Call` boundary wrapper (in Python's ctypes glue) must catch `RError`
    and translate it into a Python exception via a `try { ... } catch (const RError &e)
     { /* set error flag */ }` block.

**Category E -- R Interpreter Item** (e.g., `eval`, `findVar`, `findVarInFrame`, `
    R_getVar`, `install`, `R_GlobalEnv`, `R_NaString`, `ENVSXP`, `LANGSXP`):
- Apply Invariant 3 strictly.
- Mark the item as an R Interpreter Item in the guide header.
- Provide the full C++ function pointer stub: global function pointer variable, a C-
    linkage registration function callable from Python, and the stub body that calls
    the pointer (or throws `RError` if the pointer has not been registered).
- Provide the complete Python-side `ctypes` snippet showing type declaration, callback
    implementation, and registration call.
- State which code paths in the package require this item and which paths do not (so
    the user knows exactly what is blocked and what is free).

## Output Format Schema

You must output the final fake header guide strictly in Markdown format. Use the exact
    headings and structure outlined below:

---

### 1. Overview of `{external_item}` in R API

*Provide a 2-3 sentence definition of the external item: what it is, its role in R's C
    API, its typical inputs and outputs, and -- if it falls under Invariant 3 -- a
    clear statement that it is an **R Interpreter Item**.*

---

### 2. Contextual Usage Analysis

*Summarize the data extracted from Step 1. List: the C types of all arguments and
    return values seen in context, which other R API items appear alongside this one (e.
    g., `PROTECT`, `ALLOC`, adjacent accessor calls), and the distinct implementation
    patterns present across the CSV rows. Identify which patterns share the same fake
    strategy and which require separate treatment.*

---

### 3. Fake C++ Implementation Strategy

*State which implementation Category (A-E) this item belongs to. Then describe:*
- *The specific fake mechanism chosen and why it satisfies the invariants.*
- *For Category C items: which allocations go to the arena vs. the heap, and how the `
    ArenaFrame` guard interacts with the function that calls this item.*
- *For Category D items: the `RError` exception class, how `vsnprintf` is used to
    format variadic arguments, and what the `.Call` boundary `try/catch` looks like.*
- *For Category E items: why a full fake is impossible, what the function pointer
    bridge achieves, and the exact boundary at which Python registers the callback.*
- *Any `#define` aliases from the original header (e.g., `#define error Rf_error`, `#
    define PROTECT(s) Rf_protect(s)`) that must be preserved in the fake header so that
     the original source files compile unchanged.*
```

```
---

### 4. Fake Implementation Examples

*For each distinct usage pattern identified in Step 2, create a subsection formatted
    exactly as follows:*

#### Pattern: [Brief Description of Pattern, e.g., Allocate 1D Integer Vector]
- **Locations:** [List `file_name:line_number` for every CSV row belonging to this
    pattern]
- **Original R API Usage:** [A representative code snippet from the source, showing the
     original SEXP-based call]
- **C++ Fake Implementation:** [A complete, syntax-highlighted C++ snippet providing
    the fake definition for this item, in the context of this pattern. Include the `
    ArenaFrame` guard if the pattern involves arena memory, and the `try/catch`
    boundary wrapper if the pattern involves error throwing.]
- **Arena / Memory Notes:** [Only if this pattern allocates memory. Specify: does this
    allocation go to the arena or to the heap? When is it freed? What happens if the
    allocation fails?]
- **Python Interop Notes:** [Only for Category E items. Show the full ctypes
    registration snippet and the C++ function pointer stub. Explain which Python-side
    data structures map to which C arguments.]
- **Explanation:** [Detail the mechanical changes: what the fake header defines, what
    `#define` aliases are preserved, and why the original source file compiles without
    modification.]

---

### 5. Integration Requirements

*List every other `external_item` whose fake guide must be included before this one in
    the final fake header (i.e., compile-order dependencies). For each dependency,
    state which specific definition from that guide is needed here (e.g., "`SEXP` guide
     required -- provides the `SEXPREC` struct used in `sexp->data`"). If there are no
    dependencies, state "None."*

---

## Output Saving Instructions
- **File Naming:** Save the generated guide as `{external_item}.md` (e.g., `allocVector
    .md`, `INTSXP.md`, `error.md`).
- **Output Directory:** Save the file to the user-specified output directory. If the
    user did not explicitly provide one, create and use the default directory path `r-
    extern-analysis/fake_guides/`.
```

# D Phase 4: Fake Header Generation

## D.1 Skill: /generate-r-extern-fake-headers

```
---
name: generate-r-extern-fake-headers
```

````
description: Generates fake C++ header files for all R C API external items listed in a
     CSV table, using pre-generated Markdown fake guides as implementation blueprints,
    and assembles a master entry-point header that replaces R.h and Rinternals.h for
    standalone builds.

Generate Fake C++ Header Files for R C API External Items

Description

Given four inputs -- a C source `base_folder`, a `csv_file` mapping all R external
    items that need faking, a `guides_folder` containing the pre-generated Markdown
    fake implementation guides (one per `external_item`), and an `output_folder` for
    the resulting `.h` files -- your task is to:

1. Parse the CSV to determine the ordered list of unique `external_item` values.
2. Sequentially invoke the `@generate-r-extern-fake-header` agent for each item to
    produce a corresponding `{external_item}.h` file in the `output_folder`.
3. After all individual header files have been generated, write a master entry-point
    header (`fake_R.h`) in the `output_folder` that `#include`s every generated header
    in the correct dependency order, so that package C source files need only add one
    include directive to compile without `libR.so`.

Expected CSV Schema

The `csv_file` must adhere to the following example schema:
```csv
external_item,header_file,category,file_name,line_number,context_statement
DL_FUNC,R_ext/Rdynload.h,type,init.c,12,"static const R_CallMethodDef CallEntries[] = {
     {""init_rpcallback"", (DL_FUNC) &init_rpcallback, 5}, {""rpart"", (DL_FUNC) &rpart
    , 11}, {""xpred"", (DL_FUNC) &xpred, 15}, {""rpartexp2"", (DL_FUNC) &rpartexp2, 2},
     {""pred_rpart"", (DL_FUNC) &pred_rpart, 12}, {NULL, NULL, 0} };"
DllInfo,R_ext/Rdynload.h,type,init.c,23,R_init_rpart(DllInfo * dll)
INTSXP,Rinternals.h,type,pred_rpart.c,139,"SEXP where = PROTECT(allocVector(INTSXP, n))
    ;"
INTSXP,Rinternals.h,type,rpart.c,194,"which3 = PROTECT(allocVector(INTSXP, n));"
INTSXP,Rinternals.h,type,rpart.c,278,"inode3 = PROTECT(allocMatrix(INTSXP, nodecount,
    6));"
INTSXP,Rinternals.h,type,rpart.c,285,"isplit3 = PROTECT(allocMatrix(INTSXP, splitcount,
     3));"
INTSXP,Rinternals.h,type,rpart.c,293,"csplit3 = PROTECT(allocMatrix(INTSXP, catcount,
    maxcat));"
INTSXP,Rinternals.h,type,rpartexp2.c,47,"SEXP keep = PROTECT(allocVector(INTSXP, n));"
REALSXP,Rinternals.h,type,rpart.c,241,"cptable3 = PROTECT(allocMatrix(REALSXP, xvals >
    1 ? 5 : 3, rp.num_unique_cp));"
REALSXP,Rinternals.h,type,rpart.c,261,"dnode3 = PROTECT(allocMatrix(REALSXP, nodecount,
     (3 + rp.num_resp)));"
REALSXP,Rinternals.h,type,rpart.c,269,"dsplit3 = PROTECT(allocMatrix(REALSXP,
    splitcount, 3));"
REALSXP,Rinternals.h,type,xpred.c,209,"predict2 = PROTECT(allocVector(REALSXP, n * ncp
    * nresp));"
```

Execution Steps
````

```
### Step 1: Identify and Order Unique External Items

Read and parse the ‘csv_file‘. Extract the definitive ordered list of all unique values
     in the ‘external_item‘ column, preserving their first-occurrence order within the
    file.

### Step 2: Generate Individual Fake Header Files

Iterate sequentially through the ordered list from Step 1. Sequential execution is
    mandatory: earlier items (e.g., ‘SEXP‘, ‘SEXPTYPE‘, ‘fake_arena‘) define types and
    memory structures that later items (e.g., ‘allocVector‘, ‘INTEGER‘, ‘REAL‘, ‘error
    ‘) must reference. Each item’s generated header must exist and be readable before
    the next agent is invoked. For each unique ‘external_item‘, strictly execute the
    following sub-steps:

1. **Extract the CSV Subset:** Isolate all rows where the ‘external_item‘ column
    matches the current target. Prepend the exact CSV header row (‘external_item,
    header_file,category,file_name,line_number,context_statement‘) to form a valid,
    standalone CSV-formatted string.
2. **Invoke the Header Generation Agent:** Call the ‘@generate-r-extern-fake-header‘
    agent, passing the ‘base_folder‘, the CSV subset string, the ‘guides_folder‘, and
    the ‘output_folder‘.
3. **Non-Blocking Error Handling:** If the agent fails, times out, or produces a
    syntactically invalid header, you must:
    - Log the precise error to the console: ‘ERROR: Failed to generate fake header for
    {external_item}. Proceeding to next item.‘
    - Immediately continue to the next ‘external_item‘. Do not halt the overall batch.

### Step 3: Assemble the Master Entry-Point Header

After all individual agents have completed (including any that were skipped due to
    errors), assemble a master header file named ‘fake_R.h‘ in the ‘output_folder‘.
    This file serves as the single drop-in replacement for R’s standard includes (‘R.h
    ‘, ‘Rinternals.h‘, ‘Rdefines.h‘, ‘R_ext/Error.h‘, etc.) in the package source files.


To build ‘fake_R.h‘:

1. **Scan the output folder** for all ‘.h‘ files that were successfully generated in
    Step 2. Collect their filenames in the same order they were produced (which
    preserves the dependency-safe generation order from Step 1).
2. **Write the master header** with the following structure:
    - A file-level comment block identifying this as the fake R API entry point,
    listing the real R headers it replaces, and noting that it was auto-generated.
    - ‘#ifndef‘/‘#define‘/‘#endif‘ header guards and a ‘#pragma once‘ guard.
    - An ‘#include "fake_arena.h"‘ directive as the first include, since the arena
    underpins all memory allocation fakes. Add a comment noting that ‘fake_arena.h‘
    must be present in the same directory and is not auto-generated by this tool.
    - One ‘#include‘ line per successfully generated ‘{external_item}.h‘ file, in
    generation order.
    - A trailing comment block summarizing which original R headers are now covered and
     which ‘external_item‘ values (if any) were skipped due to agent errors.
```

````
3. **Do not include** any `{external_item}.h` that was not successfully written to disk
    ; only reference files that actually exist in the output folder.
````

D.2 Agent: `@generate-r-extern-fake-header`

````
---
name: generate-r-extern-fake-header
description: Writes a fake C++ header file for a specific R C API external item by
    translating the implementation blueprint from its pre-generated Markdown guide into
     compilable C++ code, so the original package source files build without libR.so.
---

# Generate a Fake C++ Header File for an R C API External Item

## Description

When provided with a `base_folder`, a CSV text snippet, a `guides_folder` containing
    pre-generated Markdown fake implementation guides, and an `output_folder`, your
    task is to produce a single self-contained C++ header file -- `{external_item}.h`
    -- that provides a drop-in fake implementation of the target R C API item. The
    original package C source files must be able to `#include` this header (
    transitively through `fake_R.h`) and compile without any modification and without
    linking against `libR.so`.

The input CSV will strictly adhere to the following example schema:
```csv
external_item,header_file,category,file_name,line_number,context_statement
INTSXP,Rinternals.h,type,pred_rpart.c,139,"SEXP where = PROTECT(allocVector(INTSXP, n))
    ;"
INTSXP,Rinternals.h,type,rpart.c,194,"which3 = PROTECT(allocVector(INTSXP, n));"
INTSXP,Rinternals.h,type,rpart.c,278,"inode3 = PROTECT(allocMatrix(INTSXP, nodecount,
    6));"
INTSXP,Rinternals.h,type,rpart.c,285,"isplit3 = PROTECT(allocMatrix(INTSXP, splitcount,
     3));"
INTSXP,Rinternals.h,type,rpart.c,293,"csplit3 = PROTECT(allocMatrix(INTSXP, catcount,
    maxcat));"
INTSXP,Rinternals.h,type,rpartexp2.c,47,"SEXP keep = PROTECT(allocVector(INTSXP, n));"
```

The generated header must:
1. Implement every function, macro, type, or constant that the `external_item`
    represents, exactly as specified in the corresponding Markdown guide in `
    guides_folder`.
2. Preserve every `#define` alias that the original R header defined for the item (e.g
    ., `#define error Rf_error`, `#define PROTECT(s) Rf_protect(s)`) so that the
    original source files compile without any textual changes.
3. Satisfy the three invariants stated in the Markdown guide: C++ exception-based
    errors, arena-backed memory management without source modification, and best-effort
     Python function-pointer bridges for R interpreter items.

## Execution Steps

### Step 1: Ingest Inputs and Gather Deterministic Context
````

```
- **Guide Ingestion:** Read the Markdown guide at `{guides_folder}/{external_item}.md`.
    Extract and fully understand all five sections: Overview, Contextual Usage
    Analysis, Fake C++ Implementation Strategy, Fake Implementation Examples, and
    Integration Requirements. The guide is the authoritative blueprint; the steps below
    gather additional context to ensure the generated header is correct.
- **Source File Inspection:** Iterate through each row in the CSV. Construct the
    absolute source file path from `base_folder` and the `file_name` column. Navigate
    to the `line_number` and read at least 15 lines above and below. Use these windows
    to confirm: the exact C types of arguments and return values that the item is used
    with, the scope and lifetime of surrounding variables, whether the item appears
    inside a function body or at file scope, and any adjacent R API calls that interact
    with it. If the source context reveals usage patterns not covered by the guide,
    note them explicitly in the generated header as code comments.
- **Original Header Resolution:** Read the original R header at `~/.conda/envs/r-to-
    python/lib/R/include/{header_file}` to extract the exact function signature, macro
    definition, or type layout declared there. Also check sub-headers under `~/.conda/
    envs/r-to-python/lib/R/include/R_ext/` for any transitively included definitions.
    If the header cannot be found in either `base_folder` or the conda include path `~/.
    conda/envs/r-to-python/lib/R/include/`, state this in a comment within the
    generated `.h` file and proceed with the definition from the Markdown guide. Never
    search other locations.

### Step 2: Resolve Header Dependencies

- **Read Integration Requirements:** Locate the "Integration Requirements" section (
    Section 5) of the Markdown guide. This section lists every other `external_item`
    whose fake header must be included before the current one. For each listed
    dependency, verify that the corresponding `{dependency}.h` file exists in the `
    output_folder`.
- **Inspect Existing Headers:** For each dependency that exists, read its content.
    Confirm that the types, structs, and function signatures it exposes are consistent
    with what the current item's guide expects. If a dependency is missing from `
    output_folder`, include a `// WARNING: dependency {dependency}.h not found in
    output_folder` comment in the generated header at the point of its `#include`
    directive and continue; do not abort.
- **Check for Conflicts:** Scan all existing headers in `output_folder` to ensure the
    current item does not redefine a symbol already declared there. If a conflict
    exists, emit the new definition conditionally using `#ifndef` guards rather than
    duplicating it.

### Step 3: Write the Fake Header File

Translate the Markdown guide's "Fake C++ Implementation Strategy" (Section 3) and "Fake
    Implementation Examples" (Section 4) into a complete, compilable C++ header file.
    Apply the following rules without exception:

**Rule 1 -- Use both `#ifndef` `#define` `#endif` header guards and `#pragma once`.**
You have to add traditional `#ifndef`/`#define`/`#endif` include guards to prevent
    multiple inclusions. Also, place `#pragma once` as the very first non-comment line.

**Rule 2 -- Dependency includes come first.**
After the header guards, emit one `#include "{dependency}.h"` line per dependency
    identified in Step 2, in the order listed by the guide's Integration Requirements.
```

```
    If the item is in Category C (memory allocation), also emit `#include "fake_arena.h
    "` before all other includes.

**Rule 3 -- Implement exactly what the guide specifies for the item's category.**
- **Category A (type or enum constant):** Emit the C++ `struct`, `typedef`, or `enum`
    definition. For `SEXP`/`SEXPREC`, output the full struct with `SEXPTYPE type`, `int
     length`, `int nrow`, `int ncol`, and a `union` of typed data pointers.
- **Category B (accessor macro or inline function):** Emit `inline` C++ functions, not
    macros, for type safety. The function body casts `sexp->data` or reads scalar
    fields from the `SEXPREC` struct. Preserve any `#define` alias from the original
    header beneath the `inline` definition.
- **Category C (allocation or memory function):** Implement heap-allocating functions
    (`allocVector`, `allocMatrix`, `mkChar`) as `inline` functions using `new` / `std::
    malloc`. Implement arena-delegating functions (`R_alloc`, `R_chk_calloc`, `S_alloc
    `) as `inline` functions calling `arena_alloc` / `arena_calloc` from `fake_arena.h`.
     Implement `PROTECT` and `UNPROTECT` as no-op `inline` functions. Place an `
    ArenaFrame` usage comment block showing callers exactly where to declare `
    ArenaFrame _frame;` at their function entry.
- **Category D (error, warning, or print function):** Define the `RError` struct
    exactly once (guard with `#ifndef FAKE_R_RERROR_DEFINED`). Implement `Rf_error` as
    a `[[noreturn]]` `inline` function that formats its variadic arguments into a `std
    ::string` via `vsnprintf` and throws `RError`. Implement `Rf_warning` as an `inline
    ` function that writes to `stderr` via `std::fprintf`. Implement `Rprintf` and `
    REprintf` as `inline` forwarding functions. Emit all `#define` aliases from the
    original R error header (`#define error Rf_error`, `#define warning Rf_warning`).
- **Category E (R interpreter item):** Emit the function pointer declaration as a
    global `extern "C"` variable, the registration function with C linkage, and the
    stub body that calls the pointer or throws `RError("...")` if the pointer is null.
    Wrap the entire block in a `// *** R INTERPRETER ITEM -- requires Python callback
    registration ***` comment block.

**Rule 4 -- Preserve all original R header `#define` aliases.**
After the implementation code, emit a block labelled `// --- Compatibility aliases (
    preserve original R API names) ---` containing every `#define` the original R
    header declared for this item. These ensure the unchanged package source files see
    the same names they used before (e.g., `#define allocVector Rf_allocVector` if the
    original header used that remapping).

**Rule 5 -- Emit an `ArenaFrame` usage note for any Category C header.**
After the alias block, emit a comment block titled `// --- ArenaFrame usage ---` that
    shows the exact one-liner a caller must add at the top of any `.Call`-style
    function body that calls arena-backed functions defined in this header.

**Rule 6 -- No runtime R linkage.**
The generated header must introduce zero dependencies on `libR.so`. Do not `#include`
    any original R headers. The only permitted system includes are from the c/C++
    standard library (`<stdlib.h>`, `<stddef.h>`, `<string.h>`, `<stdexcept>`, `<stdio.
    h>`, `<stdarg.h>`, `<vector>`, `<string>`, `<unordered_map>`, etc.) and from other
    fake headers already in the `output_folder`.

## Output Format Schema

The generated `.h` file must follow this exact structure, in order:
```

```
// ============================================================
// {external_item}.h -- Fake R API: {external_item}
// Original R header: {header_file}
// Category: {A|B|C|D|E} -- {category description}
// Auto-generated by generate-r-extern-fake-header agent.
// ============================================================
#pragma once

// --- Standard library includes ---
#include <...>

// --- Fake R header dependencies ---
// (from Integration Requirements in {external_item}.md)
#include "fake_arena.h"        // if Category C
#include "{dependency_1}.h"
#include "{dependency_2}.h"

// --- R Interpreter Item warning block (Category E only) ---
// *** R INTERPRETER ITEM -- {external_item} ***
// A complete fake is impossible without an R interpreter.
// Register a Python callback via {external_item}_register() before use.
// Affected code paths: [list from guide]

// --- Core fake implementation ---
// [struct / typedef / enum / inline function / function pointer stub]
// One block per usage pattern from Section 4 of the Markdown guide.
// Each block is preceded by a comment citing the guide pattern name
// and the source file locations it covers.

// --- Compatibility aliases (preserve original R API names) ---
// [#define aliases from the original R header]

// --- ArenaFrame usage (Category C only) ---
// Declare `ArenaFrame _frame;` as the first statement of any
// .Call-entry function that invokes arena-backed functions from
// this header. Example:
//   SEXP my_entry(SEXP x) { ArenaFrame _frame; ... }
```

If the `external_item` belongs to Category E (R interpreter item), the file must
    additionally contain, after the core fake implementation block, a Python-side usage
     note formatted as a multi-line C++ comment block that reproduces the full `ctypes`
     registration snippet from the Markdown guide's "Python Interop Notes" section.

Output Saving Instructions
- **File Naming:** Save the generated header as `{external_item}.h` (e.g., `allocVector
    .h`, `INTSXP.h`, `error.h`).
- **Output Directory:** Save the file to the user-specified `output_folder`. If the
    user did not explicitly provide one, create and use the default directory `
    r2py_rpart/r_fake_headers/`.

E Phase 5A: Pure-C Entry-Point Generation

E.1 Skill: /generate-r-extern-raw-entry-points

```
---
name: generate-r-extern-raw-entry-points
description: Scans R source files for .Call/.External invocations, identifies every C
    function called directly from R, and generates a pure C entry-point wrapper for
    each that replaces SEXP-based parameters and return values with plain int/double
    arrays callable from Python.
---

# Generate Pure C Entry Points for R-Callable C Functions

## Description

Given four inputs -- an `r_base_folder` containing R source files, a `c_base_folder`
    containing the package C source files, a `fake_headers_folder` containing the pre-
    generated fake R C API headers, and an `output_folder` for the resulting `.c` files
     -- your task is to:

1. Recursively scan every `.R` file in `r_base_folder` to locate all `.Call` and `.
    External` invocations and record the C function name, call type, R call expression,
     and source location for each one.
2. Identify all unique C functions that are invoked directly from R, consolidating
    multiple call sites for the same function.
3. Sequentially invoke the `@generate-r-extern-raw-entry-point` agent for each unique C
     function to produce a `{c_function}_c.c` file in `output_folder` containing a
    plain-C wrapper that exposes the function to Python without SEXP.

## Execution Steps

### Step 1: Scan R Source Files and Build the Call Site Table

Recursively scan `r_base_folder` for all files with a `.R` or `.r` extension. For every
     R file found, parse it line by line to locate every `.Call(...)` and `.External
    (...)` invocation. For each invocation found, extract and record the following four
     data points:

- **`c_function`**: The name of the C function being called. This is the first argument
     to `.Call`/`.External`. It may appear as:
  - A registered native symbol object (e.g., `C_rpart`, `C_pred_rpart`) -- strip the `
    C_` prefix to obtain the C function name.
  - A quoted string literal (e.g., `"rpart"`, `"pred_rpart"`) -- use the string content
     directly.
- **`call_type`**: Either `Call` (for `.Call(...)`) or `External` (for `.External(...)
    `).
- **`r_file`**: The path of the R file, relative to `r_base_folder`.
- **`r_line`**: The line number where the `.Call`/`.External` expression begins.
- **`r_call_expression`**: The complete `.Call`/`.External` expression, including all
    argument names or expressions passed after the function identifier. If the call
    spans multiple lines, consolidate it into a single string with internal whitespace
    collapsed.
```

````
Compile all records into an internal call site table adhering to this CSV schema:

```csv
c_function,call_type,r_file,r_line,r_call_expression
rpart,Call,rpart.R,145,".Call(C_rpart, ncat, method, opt, parms, xvals, xgrp, ymat,
    xmat, wt, ny, cost)"
xpred,Call,xpred.rpart.R,52,".Call(C_xpred, ncat, method, opt, parms, xvals, xgrp, ymat
    , xmat, wt, ny, cost, all, cp, toprisk, nresp)"
pred_rpart,Call,predict.rpart.R,89,".Call(C_pred_rpart, dimx, nnode, nsplit, dimc, nnum
    , nodes2, vnum, split2, csplit2, usesur, xdata2, xmiss2)"
rpartexp2,Call,rpart.R,256,".Call(C_rpartexp2, dtimes, eps)"
init_rpcallback,Call,rpart.R,198,".Call(C_init_rpcallback, rho, ny, nr, expr1, expr2)"
```

### Step 2: Identify Unique C Functions and Consolidate Call Sites

From the call site table built in Step 1, extract the definitive ordered list of unique
     `c_function` values, preserving the first-occurrence order across the scanned R
    files. For each unique C function, collect all rows from the table that share that
    `c_function` value into a grouped CSV subset -- multiple rows exist when the same C
     function is called from more than one R file or location.

### Step 3: Generate Entry-Point Files Sequentially

Iterate through the ordered list of unique C functions from Step 2 one at a time. For
    each function, strictly execute the following sub-steps:

1. **Prepare the CSV Subset:** Take all rows from the call site table where `c_function
    ` matches the current target. Prepend the exact header row (`c_function,call_type,
    r_file,r_line,r_call_expression`) to form a valid, standalone CSV-formatted string.
2. **Invoke the Entry-Point Agent:** Call the `@generate-r-extern-raw-entry-point`
    agent, passing the `r_base_folder`, `c_base_folder`, `fake_headers_folder`, `
    output_folder`, and the CSV subset string.
3. **Non-Blocking Error Handling:** If the agent fails, times out, or produces
    syntactically invalid C code, you must:
    - Log the precise error to the console: `ERROR: Failed to generate raw entry point
    for {c_function}. Proceeding to next function.`
    - Immediately continue to the next C function in the list. Do not halt the overall
    batch.
````

E.2 Agent: `@generate-r-extern-raw-entry-point`

````
---
name: generate-r-extern-raw-entry-point
description: Writes a pure C entry-point wrapper for a single R-callable C function,
    replacing its SEXP-based parameters and return value with plain int/double array
    arguments so the function can be called directly from Python without libR.so.
---

# Generate a Pure C Entry-Point Wrapper for an R-Callable C Function

## Description
````

```
When provided with an `r_base_folder`, a `c_base_folder`, a `fake_headers_folder`
    containing pre-generated fake R C API headers, an `output_folder`, and a CSV text
    snippet listing every R call site that invokes a specific C function, your task is
    to produce a single C source file -- `{c_function}_c.c` -- that wraps the original
    SEXP-based function in a new `{c_function}_c` entry point that Python can call
    directly via `ctypes` or `cffi` using only plain `int`, `double`, and `char`
    pointer arguments.

The input CSV will strictly adhere to the following schema:
```csv
c_function,call_type,r_file,r_line,r_call_expression
pred_rpart,Call,predict.rpart.R,89,".Call(C_pred_rpart, dimx, nnode, nsplit, dimc, nnum
    , nodes2, vnum, split2, csplit2, usesur, xdata2, xmiss2)"
pred_rpart,Call,predict.rpart.R,201,".Call(C_pred_rpart, dimx2, nnode, nsplit, dimc,
    nnum, nodes2, vnum, split2, csplit2, usesur, xdata2, xmiss2)"
```

The generated file must:
1. Implement a `{c_function}_c` function whose signature uses only plain C types -- `
    int *`, `double *`, `int` (for scalar dimensions), and `char *` (for error output)
    -- with no SEXP anywhere in the signature.
2. Internally construct fake SEXP objects from the plain input arrays, call the
    original `{c_function}` using the fake R API from `fake_headers_folder`, unpack the
     returned SEXP into the caller-provided output arrays, and propagate any `RError`
    exception as a null-terminated string via an `error_out` parameter.
3. Compile without `libR.so` by relying exclusively on the fake headers in `
    fake_headers_folder`.

Execution Steps

Step 1: Ingest Inputs and Gather Deterministic Context

- **R Call Site Inspection:** For every row in the CSV, construct the absolute R file
    path from `r_base_folder` and the `r_file` column. Navigate to the `r_line` and
    read at least 20 lines above and below. From this window, extract:
  - The declared R type of each argument passed to `.Call`/`.External` (`https://search
    .r-project.org/R/refmans/base/html/CallExternal.html`) (look for preceding `as.
    integer`, `as.double`, matrix constructions, `dim()` calls, etc.).
  - Whether each argument is a scalar, a 1-D vector, or a 2-D matrix (established by
    how R prepares the argument before passing it to `.Call`).
  - The names the R code uses for output components after the `.Call` returns (e.g., `
    out$which`, `out$cptable`), to understand the return structure.
  - Whether `call_type` is `Call` or `External`. For `.External`, note that all
    arguments arrive as a single SEXP pairlist; document this difference prominently in
     the generated file header comment but still provide the best-effort flat-array
    entry point.

- **C Function Definition Lookup:** Search `c_base_folder` recursively for the file
    that defines `{c_function}` (the function whose first token matches `SEXP\s+{
    c_function}\s*\(`). Read the complete function signature and the first 60 lines of
    the function body to determine:
  - The exact number and SEXP parameter names.
  - Which SEXP parameters are used as integer vectors (`INTEGER()`), real vectors (`
    REAL()`), integer matrices (`INTEGER()` + `nrows()`/`ncols()`), real matrices, or
```

```
    scalar integers/reals (‘asInteger()‘/‘asReal()‘).
  - The type and structure of the return SEXP (scalar, 1-D vector of a specific type,
    or a named list ‘VECSXP‘). For a list return, read far enough into the function
    body to identify every ‘SET_VECTOR_ELT‘ and ‘SET_STRING_ELT‘ call, recording the
    index, name string, and SEXP type of each component.

- **Fake Header Inspection:** Read ‘{fake_headers_folder}/fake_R.h‘ (or ‘fake_R.hpp‘)
    to confirm the ‘SEXPREC‘ struct layout and the names of the fake allocation,
    accessor, and error functions. Read the individual header files for the specific
    items used by ‘{c_function}‘ (e.g., ‘INTSXP.h‘, ‘REALSXP.h‘, ‘allocVector.h‘, ‘
    PROTECT.h‘, ‘INTEGER.h‘, ‘REAL.h‘, ‘error.h‘, ‘R_alloc.h‘). This ensures the
    construction and extraction helpers in the generated file are consistent with the
    fake header definitions already established.

### Step 2: Derive the Plain-C Signature

Using the context gathered in Step 1, produce a complete mapping from the original SEXP
     parameters and return value to their plain-C equivalents. Apply the following
    rules in order:

**Input parameter mapping:**
- SEXP used as ‘asInteger(x)‘ (scalar int) -> one ‘int‘ value parameter.
- SEXP used as ‘asReal(x)‘ (scalar double) -> one ‘double‘ value parameter.
- SEXP used as ‘INTEGER(x)‘ only, with ‘LENGTH(x)‘ but no ‘nrows‘/‘ncols‘ -> ‘int *{
    name}, int {name}_len‘.
- SEXP used as ‘REAL(x)‘ only, with ‘LENGTH(x)‘ but no ‘nrows‘/‘ncols‘ -> ‘double *{
    name}, int {name}_len‘.
- SEXP used as ‘INTEGER(x)‘ with ‘nrows(x)‘/‘ncols(x)‘ -> ‘int *{name}, int {name}_nrow
    , int {name}_ncol‘.
- SEXP used as ‘REAL(x)‘ with ‘nrows(x)‘/‘ncols(x)‘ -> ‘double *{name}, int {name}_nrow
    , int {name}_ncol‘.
- SEXP that is a Category E R interpreter item (environment, expression, language
    object) -> replace with a ‘void *{name}_callback‘ function pointer of an
    appropriate ‘typedef‘-d type, with a comment explaining the Python registration
    requirement. See the corresponding guide in ‘fake_headers_folder‘ for the pointer
    type definition.

**Return value mapping:**
- SEXP that is a 1-D integer vector -> one output parameter ‘int *{name}_out‘ (pre-
    allocated by caller) and one ‘int {name}_len‘ parameter carrying the expected
    length.
- SEXP that is a 1-D real vector -> ‘double *{name}_out‘, ‘int {name}_len‘.
- SEXP that is a named list (‘VECSXP‘) -> one pair ‘{type} *{component}_out‘ /
    dimension parameter(s) per list component, plus one ‘int *has_{optional_component}‘
     flag parameter for any component whose presence is conditional (e.g., ‘csplit‘ in
    ‘rpart‘). Derive the component types and sizes from the ‘SET_VECTOR_ELT‘ analysis
    in Step 1.
- In all cases, append two trailing parameters at the end of the signature: ‘char *
    error_out‘ (a caller-allocated buffer that receives the error message if ‘RError‘
    is thrown) and ‘int error_out_len‘ (its capacity in bytes). The presence of a non-
    empty ‘error_out‘ after the call signals an error to the caller.

Document the complete derived signature in a structured comment block at the top of the
     generated file, listing each parameter, its direction (‘in‘, ‘out‘, ‘in/out‘), its
```

```
     plain-C type, and the original SEXP parameter or return component it corresponds
    to.

### Step 3: Write the `{c_function}_c.c` File

Produce the complete C source file applying the following rules without exception:

**Rule 1 -- Includes.**
Begin with a file-level comment block (see Output Format Schema). Then emit:
```c
#include "fake_R.h"   /* or whichever master fake header exists in fake_headers_folder
    */
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
```
Do not include any original R headers. Do not include the C source file that defines `{
    c_function}`; instead use an `extern` declaration (Rule 2).

Rule 2 -- Extern declaration of the original function.
Emit the exact SEXP-based function prototype from the C source file, prefixed with `
    extern`, so the linker can resolve it from the compiled package object:
```c
extern SEXP {c_function}(SEXP param1, SEXP param2, ...);
```

Rule 3 -- Static inline SEXP construction helpers.
For each distinct plain-C -> SEXP conversion required by the input parameters (
    determined in Step 2), write a `static inline` helper function that allocates a `
    SEXPREC` on the heap, sets its `type`, `length`, `nrow`, `ncol`, and `data` fields
    using the fake header definitions, and returns the `SEXP`. The helper must **not**
    copy the data -- it must point `data` directly at the caller-supplied array, since
    the original C function only reads from these inputs. Name helpers descriptively: `
    make_int_vec`, `make_real_vec`, `make_int_mat`, `make_real_mat`, `make_int_scalar`,
     `make_real_scalar`. Each helper must be guarded against a null data pointer and
    throw `RError` (from the fake error header) if `malloc` fails.

Rule 4 -- Static inline SEXP extraction helpers.
For each distinct SEXP -> plain-C conversion required by the return value (determined
    in Step 2), write a `static inline` helper that reads from a `SEXP` using the
    accessor functions from the fake headers and copies into a caller-supplied output
    array. Name helpers descriptively: `extract_int_vec`, `extract_real_vec`, `
    extract_int_mat`, `extract_real_mat`. For list components, write one helper per
    component type, taking the list SEXP, the component index, and the output pointer.
    Each extraction helper must validate the SEXP type tag matches what is expected and
     call `Rf_error` (which throws `RError`) if there is a mismatch.

Rule 5 -- The `{c_function}_c` entry-point function.
Write the entry point with the exact plain-C signature derived in Step 2. Declare it `
    noexcept` -- `ctypes` and `cffi` operate at the C ABI level and have no knowledge
    of C++ exceptions; any exception that escapes an `extern "C"` boundary produces
    undefined behaviour (a process crash on most platforms). `noexcept` makes this
    contract explicit and lets the compiler enforce it. The function body must follow
    this strict sequence:
```

````
1. Immediately zero the `error_out` buffer: `if (error_out && error_out_len > 0)
   error_out[0] = '\0';`
2. Declare `ArenaFrame _frame;` as the very next line, before any other local
   variable. This RAII guard ensures all `R_alloc` arena memory used by `{c_function}`
    is freed when `{c_function}_c` returns.
3. Construct a `SEXP` local variable for each input parameter by calling the
   appropriate helper from Rule 3.
4. Call the original function inside a try block that catches **all** exception types
   . `RError` is the expected error path; `std::bad_alloc` can be thrown by SEXP
   helper `malloc` calls; `std::exception` covers any other standard exception; the
   catch-all `(...)` ensures nothing escapes regardless. Every catch arm writes to `
   error_out` and returns:
    ```cpp
    SEXP _result;
    try { _result = {c_function}(sexp1, sexp2, ...); }
    catch (const RError &_e) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, _e.what(), error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input SEXP wrappers */
        return;
    }
    catch (const std::bad_alloc &) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, "out of memory", error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input SEXP wrappers */
        return;
    }
    catch (const std::exception &_e) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, _e.what(), error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input SEXP wrappers */
        return;
    }
    catch (...) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, "unknown C++ exception", error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input SEXP wrappers */
        return;
    }
    ```
5. Unpack the returned `_result` SEXP into caller-supplied output arrays using the
   helpers from Rule 4. For conditional list components (e.g., `csplit`), check the
   list length and set the corresponding `has_*` flag.
6. Free all heap-allocated `SEXPREC` wrapper structs created by the Rule 3 helpers (
   the underlying data arrays are **not** freed -- the caller owns them). Free the
   returned `_result` SEXP and any sub-SEXPs it contains.
````

````
The Python caller checks `error_out` after every call and raises a Python exception if
    it is non-empty:
```python
error_buf = ctypes.create_string_buffer(1024)
lib.{c_function}_c(..., error_buf, 1024)
if error_buf.value:
    raise RuntimeError(error_buf.value.decode())
```
This is the only safe error-propagation channel across the C ABI boundary when using `
    ctypes`. If native Python exception propagation is required in the future, the
    entry-point file should be ported to `pybind11` or `Cython`, which understand C++
    exceptions and can translate them automatically.

**Rule 6 -- `extern "C"` linkage guard.**
Wrap the `extern` declaration, all helper functions, and the entry point in an `#ifdef
    __cplusplus` / `extern "C"` / `#endif` block so the file compiles correctly whether
     the package is built as C or C++.

**Rule 7 -- No runtime R linkage.**
The generated file must introduce zero dependencies on `libR.so`. All SEXP types and
    API functions come exclusively from the fake headers.

## Output Format Schema

The generated `{c_function}_c.c` file must follow this exact structure, in order:

```c
/* ================================================================
 * {c_function}_c.c -- Pure C entry point for {c_function}()
 *
 * Original R call form:
 *   {r_call_expression from CSV}
 *
 * Parameter map (SEXP -> plain C):
 *   {param1} ({original SEXP type}) -> {plain C type}  [in]
 *   {param2} ({original SEXP type}) -> {plain C type}  [in]
 *   ...
 *   {return component 1} ({SEXP type}) -> {plain C type}  [out]
 *   {return component 2} ({SEXP type}) -> {plain C type}  [out]
 *   error_out  char *  [out]  -- non-empty on any C++ exception
 *   error_out_len  int  [in]   -- capacity of error_out buffer
 *
 * NOTE: ctypes/cffi have no knowledge of C++ exceptions. All exceptions
 * are caught inside this file and written to error_out. The entry point
 * is declared noexcept to enforce this at the compiler level.
 * Python callers must check error_out after every call.
 *
 * Fake headers required: fake_R.h (and transitive includes)
 * Original C source:     {relative path to C file in c_base_folder}
 * R call sites:          {r_file}:{r_line} [, ...]
 * ================================================================ */

#ifdef __cplusplus
````
````

```
extern "C" {
#endif

#include "fake_R.h"
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

/* --- Original function declaration (resolved at link time) --- */
extern SEXP {c_function}(...);

/* --- Input SEXP construction helpers (static, no-copy) --- */
static inline SEXP make_int_vec(...) { ... }
static inline SEXP make_real_mat(...) { ... }
/* ... one helper per distinct input conversion pattern ... */

/* --- Output SEXP extraction helpers (static, copy-out) --- */
static inline void extract_int_vec(...) { ... }
static inline void extract_real_mat(...) { ... }
/* ... one helper per distinct output extraction pattern ... */

/* --- Entry point (noexcept: all exceptions caught internally) --- */
void {c_function}_c(
    /* inputs */
    ...,
    /* outputs */
    ...,
    /* error propagation */
    char *error_out, int error_out_len
) noexcept {
    if (error_out && error_out_len > 0) error_out[0] = '\0';
    ArenaFrame _frame;

    /* wrap inputs */
    ...

    /* call original -- catch ALL exception types; nothing may escape */
    SEXP _result;
    try { _result = {c_function}(...); }
    catch (const RError &_e) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, _e.what(), error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input wrappers */
        return;
    }
    catch (const std::bad_alloc &) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, "out of memory", error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input wrappers */
        return;
```

```
    }
    catch (const std::exception &_e) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, _e.what(), error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input wrappers */
        return;
    }
    catch (...) {
        if (error_out && error_out_len > 0) {
            strncpy(error_out, "unknown C++ exception", error_out_len - 1);
            error_out[error_out_len - 1] = '\0';
        }
        /* free input wrappers */
        return;
    }

    /* extract outputs */
    ...

    /* free wrappers */
    ...
}

/* Python caller pattern:
 *   error_buf = ctypes.create_string_buffer(1024)
 *   lib.{c_function}_c(..., error_buf, 1024)
 *   if error_buf.value:
 *       raise RuntimeError(error_buf.value.decode())
 */

#ifdef __cplusplus
} /* extern "C" */
#endif
```

```
## Output Saving Instructions
- **File Naming:** Save the generated source file as `{c_function}_c.c` (e.g., `rpart_c
    .c`, `pred_rpart_c.c`, `rpartexp2_c.c`).
- **Output Directory:** Save the file to the user-specified `output_folder`. If the
    user did not explicitly provide one, create and use the default directory `
    r2py_rpart/c_entry_points/`.
```

F Phase 5B: Fake-Header Migration

F.1 Skill: /modify-c-folder-with-fake-headers

```
---
name: modify-c-folder-with-fake-headers
description: Batch-modifies all C source and header files in a target folder to replace
     real R API includes with fake_R.h, suppress any unfakeable symbol usages by
```

````
    commenting them out, and verifies that both the modified package sources and the
    pre-generated raw entry point files compile cleanly with g++ -x c++.
---

# Modify a C Source Folder to Use Fake R Headers

## Description

This command is designed to run **immediately after** `generate-r-extern-raw-entry-
    points`. Given a `c_folder` containing the package C source files (`.c` and `.h`),
    an `entry_points_folder` containing the pure-C entry point wrappers generated by `
    generate-r-extern-raw-entry-points` (e.g., `rpart_c.c`, `pred_rpart_c.c`), and the
    absolute path to `fake_r_header` (the master `fake_R.h` generated by `generate-r-
    extern-fake-headers`), your task is to:

1. Recursively discover every `.c` and `.h` file under `c_folder`.
2. Invoke the `@modify-c-file-with-fake-headers` agent for each file to perform in-
    place modification.
3. After all files have been processed, compile every modified `.c` file from `c_folder
    ` **and** every `.c` file from `entry_points_folder` with `g++ -x c++` to verify
    correctness, and report a per-file and aggregate result.

The goal is a codebase that compiles cleanly without `libR.so`, relying exclusively on
    the fake R API headers and the C++ standard library. The entry point files are the
    primary consumers of the modified package sources -- they must be included in the
    compilation check, since a successfully modified source file that breaks its entry
    point wrapper is not a successful outcome.

## Execution Steps

### Step 1: Discover All C Files

Recursively scan `c_folder` for every file whose name ends with `.c` or `.h`. Collect
    their absolute paths into an ordered list -- sorted alphabetically. The ordering
    has no semantic significance for correctness (all modifications are independent),
    but alphabetical order produces a deterministic, reproducible discovery log.

Log the discovered file list to the console:

```
Discovered <N> file(s) in <c_folder>:
  [1] <absolute_path_1>
  [2] <absolute_path_2>
  ...
```

### Step 2: Process All Files in Parallel

Dispatch the `@modify-c-file-with-fake-headers` agent for **all files simultaneously**.
     Each agent invocation depends only on two inputs that are both fixed before this
    step begins: the file's own on-disk content and the read-only `fake_R.h` inventory.
     No agent reads another file's modified output during its own modification pass, so
     there are no inter-file ordering constraints. For each file, the agent receives
    the following sub-steps:
````

````
1. **Invoke the modification agent.** Call the `@modify-c-file-with-fake-headers` agent
    , passing:
   - `c_file`: the absolute path to the current file.
   - `fake_r_header`: the absolute path to `fake_R.h` as provided to this command.
2. **Collect the per-file report.** Record the structured summary emitted by the agent
    (headers replaced, lines commented out, unfakeable symbols found, status).
3. **Non-blocking error handling.** If the agent fails, times out, or reports a file-
    level error, log the following and immediately continue to the next file:
   ```
   ERROR: Failed to process <c_file>. Reason: <error_message>. Proceeding to next file.
   ```
   Do not abort the overall batch.

After all files have been processed, print a cumulative processing summary:

```
Processing complete: <N_ok> file(s) modified successfully, <N_warn> with warnings, <
    N_err> failed.
```

### Step 3: Verify Compilation

After Step 2 has completed for all files, verify that every relevant `.c` file compiles
     correctly as C++. The verification covers two sets of files:

- **Set A -- modified package sources:** every `.c` file discovered in Step 1 (the
    files just modified in-place from `c_folder`).
- **Set B -- entry point wrappers:** every `.c` file found directly inside `
    entry_points_folder` (non-recursive; the folder is flat by convention). These files
     were generated by `generate-r-extern-raw-entry-points` and already contain `#
    include "fake_R.h"` and `extern SEXP` forward declarations. They are not modified
    by this command, but they must compile against the now-modified package sources to
    confirm the full pipeline is consistent.

`.h` files are excluded from direct compilation in both sets -- they are checked
    transitively when their including `.c` files are compiled.

#### 3.1 Determine the Include Paths

Let `fake_headers_dir` = the **directory** containing `fake_r_header` (i.e., `dirname(
    fake_r_header)`). This directory must be passed to `g++` via `-I` so that `#include
     "fake_R.h"` and all other fake header names resolve correctly.

#### 3.2 Compile Each `.c` File Individually

For each `.c` file in **Set A**, run:

```bash
g++ -std=c++17 \
    -x c++ \
    -I"<fake_headers_dir>" \
    -I"<c_folder>" \
    -Wall \
````
````

````
    -Wno-unused-variable \
    -Wno-unused-parameter \
    -fsyntax-only \
    "<c_file>"
```

For each `.c` file in **Set B**, run:

```bash
g++ -std=c++17 \
    -x c++ \
    -I"<fake_headers_dir>" \
    -Wall \
    -Wno-unused-variable \
    -Wno-unused-parameter \
    -fsyntax-only \
    "<entry_point_file>"
```

Entry point files do not need `-I"<c_folder>"` because they reference the original
    package functions via `extern` declarations rather than by including the package's
    internal headers.

Flag explanations:
- `-std=c++17` -- required because `fake_R.h` uses `thread_local`, `inline` variables,
    and other C++17 constructs.
- `-x c++` -- instructs `g++` to treat the `.c` file as C++ source without renaming it.
- `-I"<fake_headers_dir>"` -- resolves `#include "fake_R.h"` and peer fake headers.
- `-I"<c_folder>"` -- (Set A only) resolves project-internal includes such as `"rpart.h
    "`, `"node.h"`, `"rpartproto.h"`.
- `-Wall` -- enables broad diagnostics so genuine issues surface immediately.
- `-Wno-unused-variable` and `-Wno-unused-parameter` -- suppress warnings on variables
    and parameters that are only retained because they were commented in but not
    actively used after suppression (an expected side-effect of the comment-out
    approach).
- `-fsyntax-only` -- performs full type-checking and symbol resolution without
    producing object files or invoking the linker. This is safe to run on individual
    translation units that depend on symbols from other translation units.

#### 3.3 Record and Report Results

For each compiled file, record one of the following outcomes:

| Outcome | Condition |
|---------|-----------|
| `PASS` | `g++` exits with status 0 and emits no diagnostics. |
| `WARN` | `g++` exits with status 0 but emits one or more warning lines. |
| `FAIL` | `g++` exits with a non-zero status (compilation errors present). |

If a file produces `FAIL`, capture the full `g++` stderr output and include it in the
    report. Do not abort -- continue compiling the remaining files.

After all compilations complete, print a structured verification report:
````
````

```
=======================================
 Compilation Verification Report
=======================================
 Fake headers dir   : <fake_headers_dir>
 Source folder      : <c_folder>
 Entry points folder: <entry_points_folder>
 Files checked      : <N> (<N_a> source + <N_b> entry points)
---------------------------------------
 PASS  : <N_pass> file(s)
 WARN  : <N_warn> file(s)
 FAIL  : <N_fail> file(s)
---------------------------------------
 Per-file results:

  [PASS] <file_1>
  [PASS] <file_2>
  [WARN] <file_3>
         warning: <g++ warning text>
  [FAIL] <file_4>
         error: <g++ error text, first 10 lines>
  ...
---------------------------------------
 Overall: <PASS | WARNINGS PRESENT | COMPILATION ERRORS DETECTED>
=======================================
```

3.4 Failure Triage Guidance

If any file produces ‘FAIL‘, print the following triage checklist immediately after the report:

```
Triage checklist for compilation failures:
  1. If the error is "undeclared identifier '<name>'":
       -> '<name>' is used in the C source but not defined in any fake header.
       -> Re-invoke @modify-c-file-with-fake-headers for the failing file with
         an updated fake symbol inventory, or manually add '<name>' to the
         appropriate fake header in <fake_headers_dir>.
  2. If the error is "no member named '<field>' in 'SEXPREC'":
       -> The SEXPREC struct in SEXP.h / INTSXP.h is missing a field used by
         this file. Add the field to the struct definition.
  3. If the error is "cannot initialize a variable of type 'T' with an rvalue
     of type 'void *'":
       -> A cast is missing in a fake accessor function.  The Category B inline
         in the corresponding fake header needs an explicit cast.
  4. If the error is a redefinition conflict:
       -> Two fake headers define the same symbol.  Add an #ifndef guard around
         the duplicate definition in the later-included header.
  5. If the error relates to a commented-out parameter or missing type in a
     function signature:
       -> The @modify-c-file-with-fake-headers agent violated the parameter rule.
         Re-run the agent for that file; the parameter line must never be
         commented out.
```

```
```

F.2 Agent: `@modify-c-file-with-fake-headers`

```
---
name: modify-c-file-with-fake-headers
description: Modifies a single C source or header file in-place to replace all R API
    header includes with a single fake_R.h include, validates every R external symbol
    used in the file against the fake symbol inventory, and comments out (never removes
    ) any statement that references a symbol absent from the fake headers -- while
    never commenting out function parameter declarations.
---

# Modify a C File to Use Fake R Headers

## Description

When provided with a `c_file` (absolute path to a `.c` or `.h` file) and a `
    fake_r_header` (absolute path to `fake_R.h`), your task is to modify the file **in-
    place** so it compiles correctly with `g++ -x c++` using the fake R API instead of
    `libR.so`. Three categories of change are applied:

1. **Include replacement** -- every `#include` line that references a real R API header
     is commented out and replaced by a single `#include "fake_R.h"` directive.
2. **Symbol validation** -- every R external symbol used in the file is cross-
    referenced against the complete fake symbol inventory derived from `fake_R.h` and
    all its transitively included headers.
3. **Unfakeable-symbol suppression** -- any statement whose only function is to call or
     assign a symbol that is absent from the fake headers is commented out (not deleted
    ) with an explanatory note. The one inviolable exception: **lines that are part of
    a function parameter declaration must never be commented out**, regardless of which
     symbols appear on them.

If the file contains no R API includes and no R API symbols, it is left unchanged.

### The Non-Negotiable Parameter Rule

A line is a **function parameter line** if it appears inside the parameter list of a
    function definition -- between the opening `(` that immediately follows the
    function name and the matching `)` that precedes the opening `{` of the function
    body. This applies equally to:
- Single-line signatures: `SEXP foo(SEXP x, SEXP y) {`
- Multi-line signatures:
  ```c
  SEXP foo(SEXP x,
           SEXP y,
           int n) {
  ```

Function parameter lines **must never be commented out**, even when they reference an
    unfakeable symbol. Instead, append a trailing `/* NOTE: parameter references
    unfakeable symbol '{symbol}' -- ensure fake_R.h provides a compatible type */`
    comment to that line.
```

````
This rule exists because commenting out a parameter declaration corrupts the function's
     ABI and prevents all callers from compiling -- a far worse outcome than leaving an
     unfakeable type in a parameter position.

### Recognised R API Headers

The following include patterns identify real R API headers that must be replaced. Both
    angle-bracket and quoted forms are recognised:

```
<R.h>               "R.h"
<Rinternals.h>      "Rinternals.h"
<Rdefines.h>        "Rdefines.h"
<Rversion.h>        "Rversion.h"
<Rmath.h>           "Rmath.h"
<R_ext/Rdynload.h> "R_ext/Rdynload.h"
<R_ext/Error.h>     "R_ext/Error.h"
<R_ext/Memory.h>    "R_ext/Memory.h"
<R_ext/RS.h>        "R_ext/RS.h"
<R_ext/Utils.h>     "R_ext/Utils.h"
<R_ext/Arith.h>     "R_ext/Arith.h"
<R_ext/Boolean.h>  "R_ext/Boolean.h"
<R_ext/Print.h>     "R_ext/Print.h"
```

Additionally, any `#include` whose path matches `R_ext/` or starts with `Rinternals`, `
    Rdefines`, `Rversion`, `Rmath` is treated as an R API header.

`#include <libintl.h>` is **not** an R API header and must **never** be touched, even
    when it appears inside an `#ifdef ENABLE_NLS` block alongside real R headers.

## Execution Steps

### Step 1: Build the Fake Symbol Inventory

Read `fake_r_header` (the `fake_R.h` file). Collect every `#include "..."` directive it
     contains and recursively read each referenced header from the same directory. For
    every header file read, extract the names of all symbols defined: `typedef`s, `
    struct` tags, `enum` constants, `#define` macro names, `inline` function names, and
     `extern` variable names. Accumulate all of these into a single flat set: the **
    fake symbol inventory**.

This inventory is the ground truth for Step 4. A symbol is considered **fakeable** if
    and only if its name appears in this inventory. A symbol is **unfakeable** if it
    appears in none of the fake headers despite being an R API symbol (i.e., it would
    have been provided by one of the real R headers listed above).

**Important:** The following Category E R Interpreter Items are present in the fake
    headers as function-pointer stubs. They are therefore **fakeable** for compilation
    purposes (they compile and link; at runtime they throw `RError` if no Python
    callback has been registered). Do **not** comment out their usage sites:
- `eval`, `Rf_eval`
- `findVar`, `Rf_findVar`
````

````
- ‘findVarInFrame‘, ‘Rf_findVarInFrame‘
- ‘install‘, ‘Rf_install‘
- ‘R_getVar‘, ‘compat_getVar‘

Only symbols that are **genuinely absent** from the fake symbol inventory are
    candidates for suppression in Step 4.

### Step 2: Read the Target File

Read the entire contents of ‘c_file‘ into a line-indexed buffer. Do not modify anything
     yet. Classify every line as one of the following:
- **R-include line** -- matches one of the R API header patterns from the Description.
- **Non-R-include line** -- any other ‘#include‘.
- **Preprocessor line** -- ‘#ifdef‘, ‘#endif‘, ‘#define‘, etc.
- **Function signature line** -- part of a function definition’s parameter list (apply
    the parameter rule from the Description; see also Step 3 for detection logic).
- **Code line** -- any other line inside a function body or at file scope.
- **Comment line** -- already wrapped in ‘/* */‘ or ‘//‘.

### Step 3: Replace R Header Includes

Iterate through the line buffer. For each **R-include line**:

1. **Comment it out in place.** Replace the line with:
   ‘‘‘c
   /* [FAKE_R] <original_line_content_trimmed> */
   ‘‘‘
   The original text is preserved inside the comment so the replacement is auditable
    and reversible.

2. **Mark the insertion point.** After processing all consecutive R-include lines in a
    block, record the line index immediately following the last commented-out include
    in that block as the **insertion point** for ‘fake_R.h‘. If R-include lines appear
    in more than one location in the file (e.g., one group near the top and one after
    ‘#ifdef ENABLE_NLS‘), use the position immediately after the **first** group as the
     insertion point.

3. **Insert the fake_R.h include.** At the recorded insertion point, insert the
    following new line:
   ‘‘‘c
   #include "fake_R.h"  /* replaces all R API headers above */
   ‘‘‘
   where ‘"fake_R.h"‘ is the **basename only**. The full directory path is supplied to
    the compiler via the ‘-I‘ flag during the verification step in the command; do not
    embed an absolute path in the source file.

4. **Idempotency guard.** Before inserting, scan the entire file for any existing ‘#
    include "fake_R.h"‘ or ‘#include <fake_R.h>‘ line. If one is already present (from
    a previous run of this agent), do not insert a second copy.

5. **Files with no R includes.** If the file contains no R-include lines at all, skip
    this step entirely and proceed to Step 4 without any include modifications.

### Step 4: Validate and Suppress Unfakeable Symbols
````

````
Walk through every non-comment line in the modified buffer (after Step 3). For each
    line, tokenise it and check each identifier against the fake symbol inventory built
     in Step 1. If an identifier is both an R API symbol (i.e., it would have been
    provided by one of the real R headers) and **absent** from the fake symbol
    inventory, apply the following rules:

**Rule 1 -- Detect the containing syntactic context.**

Before deciding whether to comment out the line, determine whether it is part of a
    function parameter list:
- Parse backwards from the suspicious identifier to find the most recently opened `(`
    that has not yet been closed by a matching `)`.
- Then parse backwards further to find whether that `(` is immediately preceded by an
    identifier followed by optional whitespace -- the signature of a function call or
    function definition.
- If the `(` is part of a **function definition** (i.e., it is followed eventually by
    `{` at the same brace depth, with no intervening `;`), the line is a **parameter
    line**. Apply the parameter rule: append a trailing warning comment and do not
    comment out the line.
- If the `(` is part of a **function call** or the line is a regular statement in a
    function body, the line is a **code line** and may be commented out.

**Rule 2 -- Comment out single-line statements.**

If the line is a code line that forms a complete C statement (ends with `;`, or is a `
    return` statement, or is a stand-alone expression), comment it out by wrapping the
    entire original line:
```c
/* [UNFAKEABLE: {symbol}] <original_line_content> */
```

**Rule 3 -- Comment out multi-line statements.**

If the unfakeable symbol appears on a line that is the start of a multi-line statement
    (opening expression does not end with `;` and the next line continues the
    expression), gather all lines belonging to that statement and replace them with a
    block comment:
```c
/* [UNFAKEABLE: {symbol}] begin
   <original_line_1>
   <original_line_2>
   ...
   [UNFAKEABLE: {symbol}] end */
```

**Rule 4 -- Cascade suppression for dangling references.**

If a variable is assigned only in a statement that was suppressed by Rule 2 or Rule 3,
    and the same variable is subsequently used on another line in the same scope, that
    subsequent line must also be commented out (with the annotation `/* [UNFAKEABLE:
    cascaded from {symbol}] ... */`). Apply this rule recursively until no active code
    line references a variable that has only ever been assigned in suppressed
    statements.
````

````
**Rule 5 -- Preprocessor blocks.**

If the unfakeable symbol appears inside a `#if`/`#ifdef`/`#elif` block (e.g., `#if
    R_VERSION < R_Version(4, 5, 0)`), evaluate whether the block as a whole should be
    suppressed. If the block defines a helper function or macro that itself uses only
    unfakeable items, comment out the entire block from `#if` to the matching `#endif`,
     replacing it with:
```c
/* [UNFAKEABLE: {symbol} -- entire preprocessor block suppressed]
   <original_block_content>
*/
```
If the block contains a mix of fakeable and unfakeable items, suppress only the
    specific lines that reference unfakeable symbols within the block, following Rules
    1-4.

**Rule 6 -- Static file-scope variable declarations.**

A line of the form `static T var;` where `T` is an unfakeable type is a variable
    declaration at file scope -- not a function parameter. It may be commented out. If
    doing so creates dangling references within function bodies, apply Rule 4.

### Step 5: Write the Modified File

Overwrite `c_file` in-place with the modified line buffer. The file's line count must
    be **identical** to the original: no lines are inserted without offsetting, and no
    lines are deleted -- only replaced or commented out. (The single `#include "fake_R.
    h"` line inserted in Step 3 is the only net addition; it is injected by replacing
    one blank line or by shifting subsequent lines down by exactly one position.)

> Implementation note: the simplest approach is to build the modified file as a list of
     strings, where each original line is either kept verbatim, replaced by a comment
    variant, or in the case of the insertion point, preceded by the new include line.
    The resulting file will have at most one more line than the original (the inserted
    `#include "fake_R.h"`).

### Step 6: Report Changes

After writing, print a structured summary to the console:

```
FILE: <c_file>
  R headers replaced    : <count>
    - <original_include_1>
    - <original_include_2>
    ...
  fake_R.h inserted at  : line <N> (after <last_replaced_header>)
  Unfakeable symbols    : <count>
    - <symbol>: <file>:<line_numbers> [commented out | parameter -- warned]
  Lines commented out   : <count>
  Status                : OK | WARNINGS (see above)
```
````

````
If the file required no changes (no R includes and no unfakeable symbols), print:

```
FILE: <c_file> -- no changes required
```
````

G Phase 6: R Package Structural Dependency Analysis

G.1 Skill: /analyze-r-folder-dependencies

```
---
name: analyze-r-folder-dependencies
description: Analyzes all R scripts in a folder to extract structural dependency
    information file by file, categorizing function calls into language, internal, and
    external dependencies.
---

# Analyze an R Folder for Dependencies

## Description

When provided with a target folder, your task is to recursively scan for all R scripts,
     use the `@analyze-r-file-dependencies` agent on each file individually, and save
    the results as structured JSON files.

## Execution Steps

### Step 1: Scan for R Files

Recursively scan the provided target folder and all of its subdirectories for files
    with `.R` or `.r` extensions. Compile a complete list of their absolute file paths.

### Step 2: Process and Analyze

Iterate through the list of identified R files sequentially. For each file:
1. Invoke the `@analyze-r-file-dependencies` agent against the file and record its
    final JSON output.
2. If the command fails or throws an error for a specific file, log the error to the
    console and immediately continue to the next file in your list. Do not halt the
    entire operation.

### Step 3: Save Output

For every successfully analyzed R file, save the resulting JSON output according to
    these strict rules:
* **Naming Convention:** Use the exact name of the original R script, but replace the
    `.R` or `.r` extension with `.json` (e.g., `data_cleaning.R` becomes `data_cleaning.
    json`).
* **Output Location:** If the user did not specify the output folder, save the new `.
    json` file in `structural_analysis/{folder_name}/`, where the `{folder_name}` is
    the name of the target folder. The relative path must be the same as its
    corresponding original R file.
```

G.2 Agent: `@analyze-r-file-dependencies`

```
---
name: analyze-r-file-dependencies
description: Analyzes an R script to extract structural dependency information function
     by function, categorizing calls into language, internal, and external dependencies.

---

# Analyze an R File for Dependencies

## Description

When provided with an R file (‘.R‘), your task is to parse the file, identify every
    user-defined function, and trace all function calls made within their bodies. You
    must categorize these dependencies into strict groups and output the result as a
    structured JSON object.

## Execution Steps

### Step 1: Extract Function Declarations

Scan the target R file and identify every user-defined function declared within it (e.g
    ., ‘my_func <- function(...) { ... }‘). Create a list of these function names.

### Step 2: Analyze Function Bodies and Trace Calls

For every function identified in Step 1, thoroughly analyze its execution body. Extract
     every function call made within that body and categorize it into one of the
    following three mutually exclusive lists:

* **Language Dependencies:**
    * *Definition:* Functions provided by Base R (e.g., ‘c()‘, ‘lapply()‘, ‘print()‘)
    or functions from installed CRAN/Bioconductor packages (e.g., ‘mutate()‘, ‘ggplot()
    ‘).
    * *Heuristic:* Include any function call that includes a package namespace prefix (
    e.g., ‘dplyr::select‘) or any standard function that is not defined locally.
* **Internal Dependencies:**
    * *Definition:* Functions that are defined within the exact same R file, or
    functions that are explicitly defined in other ‘.R‘ files within the same project
    directory.
* **External Dependencies:**
    * *Definition:* Foreign language interfaces (typically C, C++, or Fortran) invoked
    from within the R code.
    * *Heuristic:* Look specifically for calls to ‘.Call()‘, ‘.C()‘, ‘.Fortran()‘, ‘.
    External()‘, or ‘useDynLib()‘. Extract the name of the compiled routine being
    invoked (usually the first string argument passed to these interfaces).

### Step 3: Generate JSON Output
```

````
Construct a JSON object where each key is the name of a function identified in Step 1.
    The value for each key must be an object containing exactly three arrays of strings
    : ‘language_dependencies‘, ‘internal_dependencies‘, and ‘external_dependencies‘.
    Deduplicate all function names within these arrays.

## Output Format Schema

You must output valid JSON strictly adhering to this structure. Do not include markdown
     code blocks if the system expects raw JSON.

‘‘‘json
{
  "function_name_1": {
    "language_dependencies": ["c", "lapply", "dplyr::filter"],
    "internal_dependencies": ["helper_function_within_file"],
    "external_dependencies": [".Call(\"c_routine_name\")"]
  },
  "function_name_2": {
    "language_dependencies": ["print"],
    "internal_dependencies": ["function_name_1"],
    "external_dependencies": []
  }
}
‘‘‘
````

H Phase 8.1: Language Dependency Extraction

H.1 Skill: /extract-r-folder-language-dependencies

````
---
name: extract-r-folder-language-dependencies
description: Recursively processes a folder of R files, utilizing pre-calculated
    structural analysis JSONs to generate detailed CSV reports of language dependency
    locations.
---

# Extract Language Dependencies from an R Folder

## Description

When provided with a target directory containing R source files and a separate
    directory containing their corresponding structural analysis results (in JSON
    format), your task is to process every R script recursively. You will invoke the ‘
    @extract-r-file-language-dependencies‘ agent on each matched file pair (the ‘.R‘
    file and its corresponding ‘.json‘ analysis), and save the extracted dependency
    data strictly as ‘.csv‘ files in a mirrored output directory.

## Execution Steps

### Step 1: Scan for R Source Files
````

```
Recursively scan the provided target directory and all of its subdirectories. Compile a
     complete list of absolute file paths for every file with an `.R` or `.r` extension.


### Step 2: Map to JSON and Process

Iterate through the list of identified R files sequentially. For each R file:
1. **Locate the JSON Input:** Find the corresponding dependency analysis JSON file. It
    will be located in the provided structural analysis directory, sharing the exact
    same relative path as the R file, but with the `.R`/`.r` extension replaced by `.
    json`.
2. **Execute File-Level Agent:** Invoke the `@extract-r-file-language-dependencies`
    agent, providing it with the R file and its paired JSON file to generate the
    dependency data.
3. **Error Handling:** If the agent fails, throws an error, or if the required JSON
    file is missing, log a clear error to the console for that specific file and
    immediately proceed to the next R file in the list. Do not halt the batch execution.


### Step 3: Save Output as CSV

For every successfully analyzed file pair, save the resulting output from the agent
    according to these strict rules:
* **Data Format:** The output must be saved strictly as a CSV file.
* **Naming Convention:** Use the exact base name of the original R script, but replace
    the `.R` or `.r` extension with `.csv` (e.g., `data_cleaning.R` becomes `
    data_cleaning.csv`).
* **Output Directory:** Maintain the original directory structure. Save the `.csv` file
     into the user-specified output directory, using the exact same relative path as
    the original R file.
* **Default Output Directory:** If the user did not explicitly specify an output
    directory, create a default directory path named `language_dependency_analysis/{
    target_folder_name}/` (where `{target_folder_name}` is the base name of the target
    R folder) and mirror the relative paths there.
```

H.2 Agent: `@extract-r-file-language-dependencies`

```
---
name: extract-r-file-language-dependencies
description: Extracts detailed location and usage data for language dependencies in an
    R file, utilizing pre-calculated structural analysis JSON.
---

# Extract Language Dependencies from an R File

## Description

When provided with an R source file (`.R`) and its corresponding structural analysis
    JSON (generated by the `@analyze-r-file-dependencies` agent), your task is to
    locate every instance where those language dependencies are invoked. You must
    extract the exact line numbers and textual calls for each dependency, and output
    the results as a well-formatted, sorted CSV.
```

````
## Execution Steps

### Step 1: Ingest Inputs

Parse the provided structural analysis JSON to identify the `language_dependencies`
    associated with each user-defined function (the JSON keys).

### Step 2: Locate and Extract Invocations

Scan the raw text of the `.R` file. For every function listed in the JSON, find its
    declaration in the R file and scan its body for the specific `language_dependencies
    ` listed in its JSON array.

For every individual call to these dependencies, extract the following data points:
* **`language_dependency`**: The exact string name of the dependency as listed in the
    JSON (e.g., `c`, `dplyr::filter`).
* **`function_name`**: The name of the user-defined function (the JSON key) where this
    dependency is currently being executed.
* **`line_number`**: The exact line number in the `.R` file where the invocation of the
     dependency begins.
* **`call_body`**: The exact code snippet of the invocation, including its arguments (e
    .g., `dplyr::filter(df, column == "value")`). If the call spans multiple lines,
    capture the entire complete statement.

### Step 3: Format and Sort Output

Compile the extracted data points into a CSV format with a header row.

## Output Format Schema

You must output valid CSV data strictly adhering to this structure. Include the header
    row. Enclose `call_body` strings in double quotes to prevent formatting errors from
     commas or line breaks within the code.

```csv
language_dependency,function_name,line_number,call_body
c,my_func_1,14,"c(1, 2, 3)"
c,my_func_2,42,"c(\"A\", \"B\")"
dplyr::filter,my_func_1,15,"dplyr::filter(data, val > 0)"
lapply,my_func_1,22,"lapply(list, function(x) {
  x + 1
})"
print,my_func_2,45,"print(\"Processing complete.\")"
```
````

I Phase 8.2: Conversion Guide Generation

I.1 Skill: /generate-language-dependency-conversion-guides

```
---
name: generate-language-dependency-conversion-guides
```

````
description: Orchestrates the generation of conversion guides for translating language
    dependencies from R to Python based on a provided CSV file.
---

# Generate Language Dependency Conversion Guides

## Description

When provided with a target `base_folder` and a CSV file matching the following schema:
```csv
language_dependency,file_path,function_name,line_number,call_body
Re,all.R,bkde,78,"Re(fft(kappa*gcounts, TRUE))"
Re,all.R,bkde2D,156,"Re(fft(rp*sp, inverse = TRUE)/(P1*P2))"
Re,all.R,bkfe,232,"Re(fft(kappam*Gcounts, TRUE))"
abs,all.R,dpill,531,abs(th24Q)
any,all.R,locpoly,610,any(bandwidth <= 0)
as.double,all.R,blkest,265,as.double(x)
as.double,all.R,blkest,265,as.double(y)
as.double,all.R,blkest,267,as.double(xj)
```
Your task is to orchestrate the batch processing of this CSV. You must extract rows
    corresponding to each unique `language_dependency` into separate CSV-formatted
    strings (each including the header row), invoke the `@generate-language-dependency-
    conversion-guide` agent for each subset, and save the resulting guides into a
    specified output directory.

## Execution Steps

### Step 1: Identify Unique Dependencies

Parse the input CSV file to identify all unique values in the `language_dependency`
    column.

*Note: The CSV is pre-sorted so identical dependencies are grouped together, allowing
    for efficient sequential processing.*

### Step 2: Extract and Process Each Dependency

Iterate through the list of identified unique `language_dependency` values sequentially
    . For each unique dependency:
1. **Extract the CSV Subset:** Isolate all rows matching the current `
    language_dependency`. Prepend the standard CSV header row to this subset to create
    a valid CSV text string.
2. **Execute Conversion Agent:** Invoke the `@generate-language-dependency-conversion-
    guide` agent. Pass the `base_folder` and the extracted CSV string as inputs to
    generate the specific conversion guide.
3. **Error Handling:** If the agent fails or throws an error during generation, log a
    clear error message to the console specifying the failed `language_dependency`, and
     immediately proceed to the next dependency in your list. Do not halt the overall
    batch execution.

### Step 3: Save Output as Markdown Files
````

````
For every successfully generated conversion guide, save the agent's markdown output to
    the file system according to these strict rules:
* **Naming Convention:** Name the output file exactly `{language_dependency}.md` (e.g.,
     `Re.md`, `abs.md`).
* **Output Directory:** Save the files to the user-specified output directory. If the
    user did not explicitly provide one, create and use a default directory path named
    `language_dependency_analysis/conversion_guides/`.
````

I.2 Agent: @generate-language-dependency-conversion-guide

````
---
name: generate-language-dependency-conversion-guide
description: Generates a conversion guide for translating a specific language
    dependency from R to Python.
---

# Generate Language Dependency Conversion Guide

## Description

When provided with a target `base_folder` and a CSV text input matching the following
    schema:
```csv
language_dependency,file_path,function_name,line_number,call_body
exp,all.R,locpoly,663,"exp(seq(log(hlow), log(hupp), length.out = Q))"
exp,all.R,sdiag,768,"exp(seq(log(hlow), log(hupp), length.out = Q))"
exp,all.R,sstdiag,845,"exp(seq(log(hlow), log(hupp), length.out = Q))"
```
Your task is to generate a comprehensive conversion guide for the specified `
    language_dependency` (e.g., `exp`). The guide must include:
1. A clear explanation of the `language_dependency`'s core functionality in R.
2. A detailed analysis of its usage within the provided CSV data, including the
    surrounding context retrieved directly from the source R files.
3. A conversion guide to equivalent Python functions/methods, providing general Python
    code snippets for all scenarios in the CSV.

## Execution Steps

### Step 1: Ingest Inputs and Gather Context

- Iterate through each row in the CSV. Using the provided `base_folder`, navigate to
    the exact `file_path` and `line_number`.
- Read the target line and the surrounding code blocks. You must read enough lines to
    fully capture multi-line function calls, identify the data types of arguments being
     passed (e.g., scalars vs. vectors), and understand how the result interacts with
    adjacent logic.
- If the R dependency's behavior is complex or ambiguous, search `https://www.
    rdocumentation.org/` to confirm its official functionality, default arguments, and
    edge cases.
- If there is ambiguity about the valid types of the functions' parameters or return
    values, you can also look them up in the document `https://cran.r-project.org/web/
    packages/rpart/refman/rpart.html`.
````

```
### Step 2: Determine Python Equivalents

- **Prioritize Vectorization:** Because R inherently vectorizes operations, you must
    default to evaluating `numpy`, `scipy`, or `pandas` as the primary Python
    equivalents.
- Choose the library that best matches R's vectorized nature. For example, if
    translating R's `exp`, strongly prefer `numpy.exp()` over the standard library `
    math.exp()` unless the context explicitly proves only a scalar is being processed.
- Carefully map R arguments to Python kwargs (e.g., translating R's `seq(..., length.
    out=Q)` to Python's `np.linspace()`).
- You are free to search online documentation for the chosen Python libraries to ensure
     you are using the most efficient and idiomatic functions.

## Output Format Schema

You must output the final conversion guide strictly in well-structured Markdown format.
     Use the following outline to ensure consistency and readability:

### 1. Overview of `{language_dependency}` in R

*Provide a concise explanation of what the dependency does, its typical inputs, and
    expected outputs.*

### 2. Contextual Usage Analysis

*Summarize how the dependency is applied across the provided dataset. Mention the data
    types involved and any recurring patterns found in the source files.*

### 3. Python Conversion Strategy

*State the chosen Python library (e.g., `numpy`) and explain why it is the most robust
    equivalent for these specific usage contexts.*

### 4. Step-by-Step Conversion Examples

*For every functional-distinct usage found in the CSV, provide a subsection containing
    :*
- **Locations:** Files and Function names.
- **Original R Context:** The types of input parameters and return values, and a
    generalized code snippet for demonstration.
- **Python Equivalent:** A complete, executable Python code snippet demonstrating the
    converted logic.
- **Explanation:** A brief breakdown of the syntax translation, addressing any nuances
    like zero-based indexing, argument mapping, or library imports.
```

J Phase 9.1: Batch R-to-Python Conversion

J.1 Skill: `/convert-r-folder-to-python`

```
---
name: convert-r-folder-to-python
```

````
description: Converts all functions in all R files within a folder to Python by
    scanning recursively and sequentially invoking /convert-r-file-to-python for each
    file.
---

# Convert All R Files in a Folder to Python

## Description

Translate every R file within a target directory into Python. You will be given the
    path to the R source folder, a folder containing pre-computed JSON dependency maps
    (one per R file, produced by `/analyze-r-folder-dependencies`), a CSV dependency
    graph, and a language dependency conversion guide folder. For each R file
    discovered, you must derive its corresponding JSON map and invoke `/convert-r-file-
    to-python` to handle the per-file translation.

## Parameters

| Parameter | Required | Description |
|---|---|---|
| `r_folder` | Yes | Path to the folder containing the R source files to convert. |
| `json_folder` | Yes | Path to the folder containing the per-file JSON dependency maps
     (e.g., `structural_analysis/R/`). Each JSON file must share the same base name as
    its corresponding R file (e.g., `all.R` -> `all.json`). |
| `csv_dependency_graph` | Yes | Path to the CSV dependency graph file (e.g., `
    structural_analysis/dependency_levels.csv`) that maps functions to their inter-file
     dependency levels. |
| `language_guides_folder` | Yes | Path to the folder containing language dependency
    conversion guides (one `.md` file per R language construct). |
| `target_output_folder` | No | Root folder for all converted output. Defaults to `
    conversion_results/R/`. Each R file's functions will be written to `{
    target_output_folder}/{R_file_name}/` as per `/convert-r-file-to-python`
    conventions. |

## Execution Steps

### Step 1: Discover All R Files

- Recursively scan `r_folder` and all of its subdirectories.
- Identify every file whose name ends with `.R` or `.r`.
- Compile a complete, deterministic list of their absolute paths, sorted alphabetically
    .
- Log the discovered file list to the console:

```
Discovered <N> R file(s) in <r_folder>:
  [1] <absolute_path_1>
  [2] <absolute_path_2>
  ...
```

### Step 2: Resolve JSON Dependency Maps

For each R file discovered in Step 1:
````

````
1. Derive the expected JSON dependency map path by replacing the file's `.R` or `.r`
    extension with `.json` and looking it up within `json_folder`. Use only the base
    filename -- do not replicate subdirectory structure inside `json_folder` unless the
     JSON files are organized the same way.
   - *Example:* `{r_folder}/all.R` -> `{json_folder}/all.json`
2. Verify that the derived JSON file exists on disk.
3. If the JSON file does **not** exist, log a warning and mark that R file as **skipped
    **:
   ```
   WARNING: No JSON dependency map found for <r_file> (expected <json_path>). Skipping.
   ```
4. Retain only files for which a corresponding JSON map was successfully located. These
     form the **execution queue**.

### Step 3: Sort the Execution Queue by Dependency Level

Before executing, sort the execution queue to respect inter-file dependency order.

1. Parse the CSV dependency graph at `csv_dependency_graph`.
2. For each R file in the execution queue, match rows by the file's **base name**
    against the `r_file` column.
3. For each matching row, extract the numeric level by stripping the optional ` (leaf)`
     suffix (e.g., `"4 (leaf)"` -> `4`).
4. Compute the **maximum level** across all matched rows for each file. If a file has
    no rows in the CSV, treat its maximum level as `0`.
5. Sort the execution queue in **descending order of maximum level** (files with the
    highest-level functions -- deepest callees -- are processed first, because other
    files depend on their outputs). Break ties alphabetically.
6. Log the sorted execution order to the console:

```
Execution order (sorted by max dependency level, highest first):
  [1] <r_file_1>  (max level: <L>)
  [2] <r_file_2>  (max level: <L>)
  ...
```

### Step 4: Execute Sequential Conversions

Iterate through the **sorted** execution queue from Step 3 one file at a time. For each
     R file:

1. **Invoke `/convert-r-file-to-python`**, passing the following explicit context:
   - The absolute path to the current R source file.
   - The absolute path to the corresponding JSON dependency map (resolved in Step 2).
   - The absolute path to the CSV dependency graph (`csv_dependency_graph`).
   - The absolute path to the language dependency conversion guides folder (`
    language_guides_folder`).
   - The target output folder (`target_output_folder`). If not provided by the user,
    use the default `conversion_results/R/`.

2. **Non-blocking error handling:** If `/convert-r-file-to-python` fails, throws an
    error, or exits abnormally for a specific file, output a clear error message to the
````

```
     console and immediately proceed to the next file. Do not halt the batch:
   ‘‘‘
   ERROR: /convert-r-file-to-python failed for <r_file>. Reason: <error_message>.
    Proceeding to next file.
   ‘‘‘

### Step 5: Report Summary

After all files in the execution queue have been attempted, print a final summary:

‘‘‘
========================================
 Convert R Folder to Python -- Summary
========================================
 R source folder      : <r_folder>
 JSON dependency maps : <json_folder>
 CSV dependency graph : <csv_dependency_graph>
 Language guides      : <language_guides_folder>
 Target output folder : <target_output_folder>
----------------------------------------
 Total R files found  : <N_total>
 Skipped (no JSON)    : <N_skipped>
 Attempted            : <N_attempted>
 Succeeded            : <N_ok>
 Failed               : <N_err>
========================================
‘‘‘

If any file failed, list them explicitly so the user knows which files to retry:

‘‘‘
Failed files:
  - <r_file_1>  Reason: <error_1>
  - <r_file_2>  Reason: <error_2>
‘‘‘
```

J.2 Sub-skill: /convert-r-file-to-python

```
---
name: convert-r-file-to-python
description: Converts all functions in an R file to Python by parsing dependency graphs
     and sequentially invoking the function conversion agent.
---

# Convert an R File to Python

## Description

Translate all functions in an entire R file into Python. You will use the R source file
    , a comprehensive JSON dependency map, a CSV dependency graph, and a language
    dependency conversion guide folder. You must parse the dependencies, establish the
    correct conversion order, and invoke the ‘@convert-r-function-to-python‘ agent for
    each function.
```

```
## Execution Steps

### Step 1: Extract Function Definitions and Dependencies

* Parse the provided JSON file. The top-level keys represent all the function names
    within the target R file.
* For each identified function, extract its corresponding R code definition from the R
    source file.
* Isolate the JSON value associated with each function key to serve as its specific
    dependency map.

### Step 2: Sort Functions by Dependency Order (Leaves to Roots)

* Filter the provided dependency CSV file by the `r_file` column to isolate entries
    relevant only to the current target file.
* Sort the functions identified in Step 1 topologically, from leaves to roots. Ensure
    that child functions always precede their parent functions in the sorted execution
    list.
* *Note:* If a child function listed in the CSV is not found in the current R file's
    JSON/source, assume it has been converted elsewhere. Ignore it for the purposes of
    sorting the current file.

### Step 3: Execute Sequential Conversions

Iterate through the sorted function list from Step 2 sequentially. For each function:
1. **Invoke Conversion Agent:** Call `@convert-r-function-to-python`. Provide the
    following context:
   * The extracted R code definition.
   * The extracted JSON dependency map.
   * The folder containing language dependency conversion guides.
   * The target output folder for the specific R file. It should be `{
    target_output_folder}/{R_file_name}/`. For example, `{target_output_folder}/all.R/`.
     If the `{target_output_folder}` is not specified, the default should be `
    conversion_results/R/`.
2. **Error Handling:** If the agent fails or throws an error for a specific function,
    log a clear error message to the console. **Do not halt** the overall batch
    execution; proceed to the next function.
```

J.3 Agent: `@convert-r-function-to-python`

```
---
name: convert-r-function-to-python
description: Converts an R function to its Python equivalent using a provided JSON
    dependency map and conversion guides.
---

# Convert an R Function to Python

## Description

Rewrite a provided R function into Python. You will receive the R function code, a JSON
     file detailing its dependencies, a folder of language dependency conversion guides
```

```
    , and a target output folder. You must maintain the exact logic and functionality
    of the original code.

## Execution Steps

### Step 1: Infer Types and Define Prototype

Identify all parameters and return types by referencing the R function's CRAN
    documentation (e.g., the `rpart` reference manual `https://cran.r-project.org/web/
    packages/rpart/refman/rpart.html`).
* Create a Python function prototype using complete type annotations.
* **Never omit any types or subtypes inside type annotations.** For example, `dict[str,
     int]` or `np.ndarray[Any, np.dtype[np.float64]]` cannot be simplified as `dict` or
     `np.ndarray` if the subtypes are known. Also, if there are multiple possible types
     for a parameter, include all of them in the annotation (e.g., `str | None`).
* *Example:* `def example_function(param1: int, param2: np.ndarray[Any, np.dtype[np.
    float64]]) -> np.ndarray[Any, np.dtype[np.float64]]:`

### Step 2: Resolve Dependencies

Analyze the provided JSON dependency map. Handle each dependency category as follows:
* **Language Dependencies:** Find the corresponding `{language_dependency}.md` files (e
    .g., `min.md`, `seq.md`) in the conversion guides folder for Python translation
    strategies and examples. You don't have to follow them strictly, but they should
    guide your translation of R constructs to Python.
* **Internal Dependencies:** Locate and review the previously converted definitions in
    the output folder (`{function_name}.json`). Strictly follow their Python logic to
    ensure correct integration.
* **External Dependencies:** Call C entry points directly via the `cffi` `_lib` object
    already set up in the module. *Example: Map `.Call(C_rpart, ...)` directly to `_lib.
    rpart_c(...)`. The available C entry points are `rpart_c`, `pred_rpart_c`, `xpred_c
    `, `rpartexp2_c`, and `init_rpcallback_c`, all loaded from `_rpart_core.so`. Use
    the module-level helpers `_iptr`, `_dptr`, `_cptr`, `_int32`, and `_float64` for
    array preparation, and always check errors with `_check_error` after each call.* If
     you are not familiar with R's `.Call` interface, refer to the R Internals manual:
    `https://search.r-project.org/R/refmans/base/html/CallExternal.html`. If you are
    not familiar with `cffi` in Python, refer to the documentation: `https://cffi.
    readthedocs.io/en/latest/`. Feel free to look up the project folder `r2py_rpart`
    for the implementation details of the C entry points and examples of how to call C
    entry points from Python.

### Step 3: Rewrite Function Body

Translate the R function body into Python syntax.
* Strictly preserve the original logic and functional intent.
* Apply the patterns learned from the language conversion guides.
* Handle R-to-Python structural differences (e.g., 1-based vs. 0-based indexing, data
    structures, control flow) appropriately.

## Output Format

Output strictly valid JSON to `{function_name}.json` in the output folder. **Do not
    wrap the output in markdown code blocks.** The function body may contain multiple
    lines, and you should include every line as a separate string in the array. Follow
```

```
    the following structure strictly (even if there are no imports, you must include an
     empty array for imports):
‘‘‘json
{
  "imports": [
    "import numpy as np"
  ],
  "function_prototype": "def converted_function(param1: int, param2: np.ndarray[Any, np
    .dtype[np.float64]]) -> np.ndarray[Any, np.dtype[np.float64]]:",
  "function_body": [
    "    # Function body goes here",
    "    pass"
  ]
}
‘‘‘
```

K Phase 9.2: Package Assembly

K.1 Skill: /combine-python-functions-into-folder

```
---
name: combine-python-functions-into-folder
description: Combines all converted Python functions from the output of /convert-r-
    folder-to-python into one Python file per R source file, then audits __init__.py
    for correctness and verifies the package builds cleanly.
---

# Combine Converted Python Functions into a Folder

## Description

Given a folder that was produced by ‘/convert-r-folder-to-python‘ (containing one
    subdirectory per R source file, each holding per-function JSON files), assemble the
     per-function JSON files for every R file into a unified Python module, name each
    output file after its R counterpart (‘.R‘ -> ‘.py‘), and then audit the package’s ‘
    __init__.py‘ to ensure all public interfaces are correctly exposed. If the package
    cannot be imported or run cleanly, fix any issues before finishing.

## Parameters

| Parameter | Required | Description |
|---|---|---|
| ‘conversion_output_folder‘ | Yes | Path to the folder produced by ‘/convert-r-folder-
    to-python‘. Contains one subdirectory per R file (e.g., ‘conversion_results/R/all.R
    /‘, ‘conversion_results/R/build.R/‘). |
| ‘python_output_folder‘ | Yes | Path to the target Python package folder where the
    combined ‘.py‘ files will be written (e.g., ‘r2py_rpart/r2py_rpart/‘). |
| ‘python_package_folder‘ | No | Root of the Python package tree used for import
    resolution and ‘__init__.py‘ auditing. Defaults to ‘python_output_folder‘. |

## Execution Steps
```

````
### Step 1: Discover All Conversion Subdirectories

- Scan `conversion_output_folder` for **immediate subdirectories** (do not recurse
    deeper).
- Each subdirectory name is an R source filename (e.g., `all.R`, `build.R`, `methods.R
    `).
- A subdirectory qualifies if it contains **at least one `.json` file** directly inside
     it.
- Compile a complete, deterministic list of qualifying subdirectories, sorted
    alphabetically by name.
- Log the discovered list to the console:

```
Discovered <N> conversion subdirectory(ies) in <conversion_output_folder>:
  [1] <subdir_name_1>  (<K> JSON file(s))
  [2] <subdir_name_2>  (<K> JSON file(s))
  ...
```

If no qualifying subdirectories are found, exit immediately with:

```
ERROR: No conversion subdirectories with JSON files found in <conversion_output_folder
    >. Nothing to combine.
```

### Step 2: Derive Target Python File Names

For each qualifying subdirectory discovered in Step 1:

1. Take the subdirectory name (e.g., `all.R`).
2. Strip the `.R` or `.r` extension and append `.py` (e.g., `all.py`).
3. The target Python file path is `{python_output_folder}/{derived_name}` (e.g., `src/
    mypkg/all.py`).
4. Log the mapping:

```
Mapping:
  <subdir_name_1>  ->  <target_python_path_1>
  <subdir_name_2>  ->  <target_python_path_2>
  ...
```

### Step 3: Execute Sequential Combinations

Iterate through the sorted list from Step 1 **one subdirectory at a time**. For each
    subdirectory:

1. **Invoke `/combine-python-functions-into-file`**, passing the following explicit
    context:
   - The absolute path to the current conversion subdirectory as the **source folder**.
   - The absolute path to the derived target Python file (from Step 2) as the **target
    file**.
````

```
2. **Non-blocking error handling:** If `/combine-python-functions-into-file` fails or
    exits abnormally, output a clear error message and proceed to the next subdirectory.
     Do not halt the batch:

   ```
   ERROR: /combine-python-functions-into-file failed for <subdir_name>. Reason: <
    error_message>. Proceeding to next subdirectory.
   ```

Step 4: Audit `__init__.py`

After all combinations have been attempted, audit the `__init__.py` file located in `
    python_package_folder` (defaulting to `python_output_folder` if not specified).

Perform the following checks and apply fixes in-place:

1. **Remove stale or unnecessary imports:** Any symbol imported or re-exported in `
    __init__.py` that no longer exists in any module inside `python_package_folder`
    must be removed.
2. **Ensure all public interfaces are exposed:** For every `.py` file written in Step 3
     (excluding `__init__.py` itself), verify that each top-level public function (
    names not prefixed with `_`) is importable via the package. If any are missing from
     `__init__.py`, add the appropriate import or `__all__` entry.
3. **No duplicate entries:** Deduplicate any import lines or `__all__` entries that
    appear more than once.
4. **Preserve existing hand-written content:** Do not remove comments, conditional
    logic, or imports that are still valid and used.

Log every change made to `__init__.py`:

```
__init__.py audit:
  Removed : <symbol_or_import>  (reason: no longer exists in package)
  Added   : <symbol_or_import>  (reason: public interface missing from exports)
  ...
  No changes required.  <- (if everything was already correct)
```

Step 5: Verify the Package Builds and Runs Cleanly

After the audit:

1. **Import check:** Attempt to import the package from its directory to verify there
    are no syntax errors or broken imports. Report the result:

   ```
   Import check: PASSED  <- or FAILED: <error>
   ```

2. **If the import fails:** Diagnose the root cause by reading the error traceback
    carefully. Apply targeted fixes to the affected files (imports, missing symbols,
    typos, indentation errors). Re-run the import check after each fix. Repeat until
    the import succeeds or you have exhausted reasonable automated remediation.
```

````
3. **Report remaining issues:** If automated remediation cannot fully resolve an import
     failure, describe exactly which file and line need manual attention.

### Step 6: Report Summary

Print a final summary:

```
========================================
 Combine Python Functions into Folder -- Summary
========================================
 Conversion output folder : <conversion_output_folder>
 Python output folder     : <python_output_folder>
 Package folder           : <python_package_folder>
----------------------------------------
 Subdirectories found     : <N_total>
 Combinations attempted   : <N_attempted>
 Succeeded                : <N_ok>
 Failed                   : <N_err>
 __init__.py changes      : <N_changes>
 Import check             : PASSED / FAILED
========================================
```

If any combination step failed, list the affected subdirectories:

```
Failed subdirectories:
  - <subdir_1>  Reason: <error_1>
  - <subdir_2>  Reason: <error_2>
```
````

K.2 Sub-skill: /combine-python-functions-into-file

````
---
name: combine-python-functions-into-file
description: Combines all converted Python functions (stored as JSON) from a specified
    directory into a unified target Python file, handling import deduplication and
    project integration.
---

# Combine Converted Python Functions into a File

## Description

Given a source folder containing JSON-formatted function data and a target Python file,
     your task is to extract, format, and combine these functions into a single Python
    script. You must resolve duplicate imports, format the code syntactically, and
    integrate it safely into the target project structure.

## Execution Steps

### Step 1: Discover and Map JSON Files
````

````
* Scan the specified source folder for all files ending with `.json`.
* The filename (excluding the `.json` extension) represents the original R function
    name.

### Step 2: Extract and Format Function Definitions

For each JSON file, extract the `imports`, `function_prototype`, and `function_body`.
* **Concatenation:** Combine the `function_prototype` and the `function_body` array
    into a single multi-line string.
* **Indentation:** Ensure standard Python indentation. The `function_prototype` should
    have zero indentation, and every line within the `function_body` must be indented
    by exactly 4 spaces relative to the prototype.

*Example JSON Structure:*
```json
{
  "imports": [
    "import numpy as np",
    "from . import _KernSmooth"
  ],
  "function_prototype": "def rlbin(X: np.ndarray[np.float64], Y: np.ndarray[np.float64
    ], gpoints: np.ndarray[np.float64], truncate: bool = True) -> dict:",
  "function_body": [
    "    n = len(X)",
    "    M = len(gpoints)",
    "    return {\"xcounts\": xcnts, \"ycounts\": ycnts}"
  ]
}
```

### Step 3: Deduplicate and Sort Imports

* Aggregate all `imports` arrays from every JSON file.
* Remove any duplicate import statements.
* Sort the deduplicated imports logically:
  1. Standard library imports (e.g., `import math`).
  2. Third-party imports (e.g., `import numpy as np`).
  3. Local project imports (e.g., `from .linbin import linbin`).
* Combine the sorted imports and the formatted function definitions into one final
    multi-line string.

### Step 4: Validate and Output the Combined Program

Do not blindly overwrite the target Python file. You must ensure the generated code
    aligns with the existing project architecture:
1. **Structural Scan:** Analyze the target project's build and configuration files (e.g
    ., `pyproject.toml`, `meson.build`, `__init__.py`) to verify correct module naming
    conventions and dependency paths.
2. **Binding Corrections:** If the structural scan reveals discrepancies in how
    external C/Fortran bindings are exposed (e.g., `_KernSmooth` is built under a
    different path than what the JSON imports suggest), automatically fix the import
    paths in your combined string to match the reality of the build system.
````

```
3. **Safe Write:** Output the finalized string into the target Python file. If the file
     already contains code, integrate the new functions safely without breaking
    existing syntax or overwriting required project-level configurations.
```

L Phase 9.3: R Interpreter Item Bridge

L.1 Skill: /add-r-interpreter-items

```
---
name: add-r-interpreter-items
description: Writes C helper entry points for Category E R interpreter items, rebuilds
    the shared library, patches __init__.py with cffi callbacks, rewrites rpartcallback.
    py to use cffi exclusively, and resolves all remaining NotImplementedError stubs to
     make the Python package functionally equivalent to the R package.
---

# Add R Interpreter Items

## Description

After `/combine-python-functions-into-folder`, two layers required for full functional
    equivalence are missing:

1. **C side** -- `make_real_sexp`, `make_int_sexp`, `make_env_sexp`, and `call_install`
     are referenced in `init_rpcallback_c.c`'s caller comment but not yet exported from
     `_rpart_core.so`.
2. **Python side** -- The converted `rpartcallback.py` mixes `ctypes` (for callbacks)
    with `cffi` (for C calls), which is incompatible. All callbacks must be rewritten
    using `ffi.callback` to match the rest of the package.

This command writes the missing C file, rebuilds the library, patches `__init__.py`,
    rewrites `rpartcallback.py`, and resolves all `NotImplementedError` stubs so that
    no stub is silently reachable at runtime.

## Parameters

| Parameter | Required | Description |
|---|---|---|
| `fake_guides_folder` | Yes | Path to the fake header guides (e.g., `r_extern_analysis
    /fake_guides/`). Provides Python Interop Notes for each Category E item. |
| `c_entry_points_folder` | Yes | Folder containing existing C entry point files (e.g.,
     `r2py_rpart/c_entry_points/`). The new helper file is written here. |
| `python_package_folder` | Yes | Python package folder containing `__init__.py` and
    the converted `.py` modules (e.g., `r2py_rpart/r2py_rpart/`). |
| `r2py_rpart_root` | Yes | Root of the installable Python package (e.g., `r2py_rpart
    /`), used for building via `pip install --no-build-isolation .`. |

## Execution Steps

### Step 1: Identify Inputs
```

````
1. **Category E guides** -- In `fake_guides_folder`, find guides whose **opening
    blockquote (line 3)** begins with `**R Interpreter Item.`** (strict prefix match).
    Exclude any guide whose opening blockquote contains `best-effort fakeable without a
     Python function pointer`. For rpart this yields `eval.md`, `findVar.md`, `
    findVarInFrame.md`. (`install.md` is excluded -- its built-in C++ hash-map requires
     no Python bridge in normal use.)

2. **Existing entry points** -- Collect the full text of `init_rpcallback_c.c` from `
    c_entry_points_folder` (contains the caller-comment protocol for `make_real_sexp`
    etc.) and one representative existing entry point (e.g., `rpart_c.c`) for style
    reference.

3. **Existing package files** -- Collect the full text of `__init__.py` and `
    rpartcallback.py` from `python_package_folder`. Also grep every `.py` file in `
    python_package_folder` for `NotImplementedError` and `ctypes` to identify all stubs
     and FFI mismatches.

Log a discovery summary:
```
Category E guides found : eval.md, findVar.md, findVarInFrame.md
Files to patch          : __init__.py, rpartcallback.py
NotImplementedError hits: <N> across <files>
ctypes usage hits       : <N> across <files>
```

### Step 2: Invoke the Agent

Invoke the `@add-r-interpreter-items` agent exactly once, providing all of the
    following as context:

- The three Category E guide texts (full content of `eval.md`, `findVar.md`, `
    findVarInFrame.md`)
- The full text of `init_rpcallback_c.c`
- The full text of one existing entry point file (style reference)
- The full text of `fake_R.h` (located in `r2py_rpart/r_fake_headers/` or adjacent)
- The full text of `__init__.py`
- The full text of `rpartcallback.py` (converted module in `python_package_folder`)
- The grep results for `NotImplementedError` and `ctypes` across all `.py` files
- The paths: `c_entry_points_folder`, `python_package_folder`, `r2py_rpart_root`

The agent handles all file writes, the build step, and all verification. Do not proceed
     past the agent invocation until it returns.

### Step 3: Report Summary

Print the agent's reported results in the following format:

```
========================================
 Add R Interpreter Items -- Summary
========================================
 C file written          : c_entry_points/interpreter_helpers_c.c
 meson.build updated     : YES / NO
 Build                   : PASSED / FAILED
````
````

```
 New symbols verified    : make_real_sexp, make_int_sexp,
                           make_env_sexp, call_install
 __init__.py patched     : YES / NO  (<N> additions)
 rpartcallback.py fixed  : YES / NO  (ctypes -> cffi)
 NotImplementedError stubs resolved : <N>
 NotImplementedError stubs remaining (intentional) : <N>
 Import check            : PASSED / FAILED
========================================
```

```
If the build or import check failed, print the full error output and do not mark the
    step as succeeded. If any `NotImplementedError` stubs remain, each must be listed
    with its file, function, and a one-line explanation of why it is intentional (e.g.,
     interactive terminal function with no Python equivalent).
```

L.2 Agent: `@add-r-interpreter-items`

```
---
name: add-r-interpreter-items
description: Writes the C interpreter-item helper entry point file, updates meson.build
    , rebuilds _rpart_core.so, patches __init__.py with cffi-based callbacks and ffi.
    cdef declarations, rewrites rpartcallback.py to use cffi exclusively, and resolves
    all NotImplementedError stubs to make the Python package functionally equivalent to
     the R package.
---

# Add R Interpreter Items

## Description

You receive:
- Full text of `eval.md`, `findVar.md`, `findVarInFrame.md` (Category E fake guides,
    especially their Section 4 Python Interop Notes)
- Full text of `init_rpcallback_c.c` (contains the caller-comment protocol for C
    helpers)
- Full text of one existing entry point file (style reference)
- Full text of `fake_R.h` (defines SEXPREC struct layout and SEXPTYPE constants)
- Full text of `__init__.py` (cffi setup, existing `ffi.cdef`, existing helpers)
- Full text of `rpartcallback.py` (currently broken: mixes ctypes callbacks with cffi C
     calls)
- Grep results for `NotImplementedError` and `ctypes` across all `.py` files
- Paths: `c_entry_points_folder`, `python_package_folder`, `r2py_rpart_root`

Your task is to close both the C and Python gaps so the package is fully functional.

## Execution Steps

### Step 1: Write `interpreter_helpers_c.c`

Read `fake_R.h` to confirm the exact `SEXPREC` field order (`type`, `length`, `nrow`, `
    ncol`, `data`). Read one existing entry point file to match the include order,
    extern-C bracketing, and noexcept style.
```

````
Write `{c_entry_points_folder}/interpreter_helpers_c.c` exporting four functions. The
    file must follow the exact same structure as the other entry points: include `
    fake_R.h` before the `extern "C"` block, wrap all exported functions in `extern "C"
     { ... }`.

**`void *make_real_sexp(void *data, int n)`**
Heap-allocates a `SEXPREC` (via `malloc`) with `type=REALSXP`, `length=n`, `nrow=n`, `
    ncol=1`, `data=data`. The data buffer is owned by the caller and is not copied.
    Returns the SEXPREC pointer cast to `void *`. On malloc failure, writes to a `
    static thread_local char[256]` error buffer and returns `nullptr`; the buffer is
    exposed via a companion `const char *get_make_sexp_error()` function. Does not use
    `RError` (this function is called from Python, not from within the C evaluation
    loop).

**`void *make_int_sexp(void *data, int n)`**
Identical to `make_real_sexp` but with `type=INTSXP`.

**`void *make_env_sexp(void)`**
Allocates a minimal `SEXPREC` with `type=ENVSXP`, `length=0`, `nrow=0`, `ncol=0`, `data
    =nullptr`. The returned pointer is a stable unique identity key for use as a `rho`
    handle in the frame registry. On malloc failure, writes to the same error buffer
    and returns `nullptr`.

**`void *call_install(const char *name)`**
Calls `Rf_install(name)` and returns the resulting SYMSXP pointer cast to `void *`.
    This exposes the built-in C++ symbol cache to Python so that Python can pre-
    populate the frame registry with the exact same pointer keys that `install()` will
    produce inside `init_rpcallback`. Safe to call from Python at any time after
    library load; does not require `ArenaFrame`.

Also add `const char *get_make_sexp_error(void)` that returns the static error buffer,
    so Python can check for allocation failures.

### Step 2: Update `meson.build`

Open `meson.build`. Find the `rpart_src = files(...)` list. Add the line:
```
  'c_entry_points/interpreter_helpers_c.c',
```
after the existing `c_entry_points/init_rpcallback_c.c` entry. Do not change any other
    part of the file.

### Step 3: Build and Verify

Run from `r2py_rpart_root`:
```bash
pip install --no-build-isolation .
```

If the build fails, read the compiler error output, fix `interpreter_helpers_c.c`, and
    retry. Do not proceed until the build passes.

After a successful build, verify all five new symbols are exported:
```bash
````
````

```
nm -D $(python -c "import r2py_rpart, os; print(os.path.dirname(r2py_rpart.__file__))")
    /_rpart_core.so \
  | grep -E "make_real_sexp|make_int_sexp|make_env_sexp|call_install|
    get_make_sexp_error"
```
All five must appear. If any are missing, add `-fkeep-inline-functions` to the compile args for those symbols or restructure the function to have external linkage.

Step 4: Patch `__init__.py` -- `ffi.cdef` Additions

Append the following block inside the existing `ffi.cdef("""...""")` call, immediately before the closing `""")`. Do not modify any existing declaration:

```c
/* ---- Interpreter-item helpers (Category E support) ---------------------- */
void *make_real_sexp(void *data, int n);
void *make_int_sexp(void *data, int n);
void *make_env_sexp(void);
void *call_install(const char *name);
const char *get_make_sexp_error(void);
```

Step 5: Patch `__init__.py` -- Module-Level Callback Infrastructure

Insert the following block into `__init__.py` immediately after the `_lib = ffi.dlopen(_find_lib())` line. This registers the `install`, `findVarInFrame`, and `findVar` callbacks at module load time. The `eval` callback is intentionally excluded here -- it is registered per-call inside `rpartcallback()` because it captures user-supplied split functions.

```python
# ---------------------------------------------------------------------------
# Interpreter-item callback infrastructure (Category E, method=4 support)
# ---------------------------------------------------------------------------

# Symbol intern cache: name (str) -> integer address of interned SYMSXP.
# Populated by _py_install; used to pre-populate _frame_registry.
_symbol_handles: dict[str, int] = {}
_symbol_bufs: dict[str, Any] = {}     # keeps ffi buffer objects alive

# Frame registry: rho_ptr (int) -> {sym_ptr (int) -> sexp_ptr (int)}.
# Populated by rpartcallback() before each init_rpcallback_c call.
_frame_registry: dict[int, dict[int, int]] = {}

# Persistent storage for module-level cffi callbacks (prevents GC).
_interp_callbacks: list[Any] = []

# Sentinel value returned by findVarInFrame when a variable is not found.
_R_UNBOUND: int = int(ffi.cast("uintptr_t", _lib.get_R_UnboundValue()))


@ffi.callback("void *(const char *)")
def _install_cb(name_bytes: Any) -> Any:
    name = ffi.string(name_bytes).decode()
```

````
    if name not in _symbol_handles:
        buf = ffi.new("char[1]")
        _symbol_bufs[name] = buf
        _symbol_handles[name] = int(ffi.cast("uintptr_t", buf))
    return ffi.cast("void *", _symbol_handles[name])


@ffi.callback("void *(void *, void *)")
def _findVarInFrame_cb(rho: Any, sym: Any) -> Any:
    rho_key = int(ffi.cast("uintptr_t", rho))
    sym_key = int(ffi.cast("uintptr_t", sym))
    val = _frame_registry.get(rho_key, {}).get(sym_key)
    return ffi.cast("void *", val) if val is not None else ffi.cast("void *",
    _R_UNBOUND)


@ffi.callback("void *(void *, void *)")
def _findVar_cb(sym: Any, rho: Any) -> Any:
    # The inherits=TRUE path through compat_getVar is dead code for all
    # standard rpart methods; this stub returns R_UnboundValue safely.
    return ffi.cast("void *", _R_UNBOUND)


_lib.register_install_fn(_install_cb)
_lib.register_findVarInFrame_fn(_findVarInFrame_cb)
_lib.register_findVar_fn(_findVar_cb)

_interp_callbacks.extend([_install_cb, _findVarInFrame_cb, _findVar_cb])
```

### Step 6: Rewrite `rpartcallback.py`

The existing converted `rpartcallback.py` is broken because it uses `ctypes.CFUNCTYPE`
    for callback creation while the library is loaded via `cffi`. Rewrite the entire
    file so it uses only `cffi`.

The rewritten function must preserve all input validation and all `eval1`/`eval2`
    computation logic from the existing conversion -- those are correct. Replace only
    the C-interface section (Steps 1-9 in the existing comments) with cffi equivalents
    following the rules below.

**Replace ctypes constructs with cffi equivalents:**

| Broken pattern (ctypes) | Correct pattern (cffi) |
|---|---|
| `ctypes.CFUNCTYPE(ret, *args)` | `ffi.callback("ret (args)")` |
| `ctypes.cast(x, ctypes.c_void_p).value` | `int(ffi.cast("uintptr_t", x))` |
| `_SEXPREC(ctypes.Structure)` local struct | `_lib.make_real_sexp(...)` / `_lib.
    make_int_sexp(...)` |
| `_lib.register_X_fn.restype = None` (ctypes API on cffi lib) | Remove -- cffi does
    not use `.restype` |
| `_lib.register_X_fn.argtypes = [...]` (ctypes API on cffi lib) | Remove -- cffi
    resolves from `ffi.cdef` |
| `_lib.register_X_fn(cb)` | `_lib.register_X_fn(cb)` (valid in cffi too, keep) |
````
````

````
**SEXP construction:** Use the new C helpers instead of local ctypes structs:
```python
sexp_y = _lib.make_real_sexp(ffi.cast("void *", rho["yback"].ctypes.data), rho["yback
    "].size)
sexp_w = _lib.make_real_sexp(ffi.cast("void *", rho["wback"].ctypes.data), rho["wback
    "].size)
sexp_x = _lib.make_real_sexp(ffi.cast("void *", rho["xback"].ctypes.data), rho["xback
    "].size)
sexp_n = _lib.make_int_sexp(ffi.cast("void *", rho["nback"].ctypes.data), rho["nback"].
    size)
```
After each call, check `ffi.string(_lib.get_make_sexp_error())` and raise `RuntimeError
    ` if non-empty.

**Symbol pointers:** Use the module-level `_symbol_handles` dict (already populated by
    `_install_cb`) rather than a local `_py_install`:
```python
# Trigger install() for each name so _symbol_handles is populated.
for name in (b"yback", b"wback", b"xback", b"nback"):
    _lib.call_install(name)
```
Then use `_symbol_handles["yback"]` etc. as the frame registry keys.

**Rho handle:** Use `make_env_sexp()` instead of a ctypes 1-byte placeholder:
```python
_rho_sexp = _lib.make_env_sexp()
_rho_ptr  = int(ffi.cast("uintptr_t", _rho_sexp))
```

**Frame registry population:** Write directly to `_frame_registry` (module-level dict
    imported from `__init__.py`, or accessed as `from r2py_rpart import _frame_registry
    `):
```python
_frame_registry[_rho_ptr] = {
    _symbol_handles["yback"]: int(ffi.cast("uintptr_t", sexp_y)),
    _symbol_handles["wback"]: int(ffi.cast("uintptr_t", sexp_w)),
    _symbol_handles["xback"]: int(ffi.cast("uintptr_t", sexp_x)),
    _symbol_handles["nback"]: int(ffi.cast("uintptr_t", sexp_n)),
}
```

**Expr handles:** Use `ffi.new("char[1]")` instead of ctypes 1-byte buffers:
```python
_expr1_buf = ffi.new("char[1]")
_expr2_buf = ffi.new("char[1]")
_expr1_ptr = int(ffi.cast("uintptr_t", _expr1_buf))
_expr2_ptr = int(ffi.cast("uintptr_t", _expr2_buf))
```

**Eval callback:** Define using `ffi.callback`. The callback must catch all Python
    exceptions and return `ffi.NULL` on failure (cffi will not propagate Python
    exceptions across the C boundary; returning NULL causes the C-side `isReal(NULL)`
    check to fail with a readable `RError`):
````

```python
_eval_result_keeper: list[Any] = []

@ffi.callback("void *(void *, void *)")
def _eval_cb(expr: Any, rho: Any) -> Any:
    try:
        expr_p = int(ffi.cast("uintptr_t", expr))
        if expr_p == _expr2_ptr:
            result = eval2()
        else:
            result = eval1()
        result = np.ascontiguousarray(result, dtype=np.float64)
        sexp = _lib.make_real_sexp(ffi.cast("void *", result.ctypes.data), result.size)
        _eval_result_keeper.append(result)   # keep numpy array alive
        return sexp
    except Exception:
        return ffi.NULL

_lib.register_eval_fn(_eval_cb)
```

Keep-alive: Store all cffi objects that must outlive the function call in `rho["_cffi_keep_alive"]`:
```python
rho["_cffi_keep_alive"] = [
    _eval_cb, _expr1_buf, _expr2_buf,
    _rho_sexp, sexp_y, sexp_w, sexp_x, sexp_n,
    _eval_result_keeper,
]
```
This replaces the `_ctypes_keep_alive` list in the existing conversion.

Return value: Keep the same return structure as the existing conversion:
```python
return {"eval1": eval1, "eval2": eval2, "rho": rho}
```

Step 7: Resolve All Remaining `NotImplementedError` Stubs

Scan every `.py` file in `python_package_folder` for `NotImplementedError`. For each occurrence, classify it and act:

Class A -- Stubs caused by missing interpreter-item wiring (must be fixed):
Any stub that says something like "C_init_rpcallback not available", "callbacks not registered", or similar. These should not exist after Step 6, but verify and fix any that remain.

Class B -- `UseMethod` dispatch stubs (must be fixed):
Stubs in `post.py`, `prune.py`, and `meanvar.py` (or wherever `UseMethod` was converted to `NotImplementedError`). Replace each with a direct dispatch based on `tree.get("_rpart_class")`:
```python
def post(tree: dict, *args, **kwargs):
    if tree.get("_rpart_class") == "rpart":
```

````
        return post_rpart(tree, *args, **kwargs)
    raise TypeError(f"no applicable method for 'post' on object of class "
                    f"'{tree.get(\"_rpart_class\", type(tree).__name__)}'")
```
Apply the same pattern to `prune` and `meanvar`.

**Class C -- Interactive or environment-specific stubs (annotate, do not silence):**
Stubs in `snip_rpart_mouse.py` (uses R's `identify()` for interactive terminal tree
    pruning -- no Python equivalent). Keep `NotImplementedError` but ensure the message
     is specific:
```python
raise NotImplementedError(
    "snip.rpart.mouse requires interactive terminal graphics (R's identify()); "
    "use snip.rpart() with explicit node indices instead."
)
```

**Class D -- Formula/model.frame stubs (annotate precisely):**
Stubs where `eval.parent(model.frame(...))` was replaced with `NotImplementedError`.
    Keep the stub but ensure the message names the specific R function and suggests the
     Python alternative:
```python
raise NotImplementedError(
    "model.frame.rpart requires a formula parser (R's model.frame/eval.parent); "
    "pass a pre-built numpy matrix as 'data' to rpart() instead of a formula string."
)
```

After processing all stubs, verify: every remaining `NotImplementedError` has a non-
    empty, specific string message. A bare `raise NotImplementedError` with no message
    is not acceptable.

### Step 8: Verify

1. **Import check:**
   ```bash
   python -c "import r2py_rpart; print('OK')"
   ```
   Must print `OK` with no exception.

2. **Callback registration check:**
   ```bash
   python -c "import r2py_rpart; assert len(r2py_rpart._interp_callbacks) == 3,
    r2py_rpart._interp_callbacks"
   ```
   Must pass (three module-level callbacks registered: install, findVarInFrame, findVar
    ).

3. **No silent stubs:**
   ```bash
   grep -rn "raise NotImplementedError()" {python_package_folder}
   ```
   Must return no output. Every `NotImplementedError` must have a message.
````
````

````
4. **No ctypes usage:**
   ```bash
   grep -rn "import ctypes\|ctypes\." {python_package_folder}
   ```
   Must return no output. All FFI must go through cffi.

Report pass/fail for each check. If any check fails, diagnose and fix before reporting
    completion.
````

M Phase 11: R Test-Suite Conversion

M.1 Skill: `/convert-r-tests-to-python`

````
---
name: convert-r-tests-to-python
description: Batch converts all R test files in a specified folder to Python by
    invoking the convert-r-test-to-python agent.
---

# Convert all R Test Files in a Folder to Python

## Description

Translate an entire directory of R test files into Python. You will require the paths
    to the R test source folder, the source R library folder, and the corresponding
    Python library folder. For every R test file discovered, you must invoke the `
    @convert-r-test-to-python` agent to handle the individual translation.

## Execution Steps

### Step 1: Discover All R Test Files

- Scan the provided R test source folder recursively.
- Identify and compile a list of all R scripts (files ending with `.R` or `.r`). Pay
    special attention to standard test naming conventions (e.g., files starting with `
    test-` or `test_`).

### Step 2: Execute Conversions Iteratively

- Iterate through the compiled list of R files from Step 1 sequentially.
- For each file, invoke the conversion agent by calling the `@convert-r-test-to-python`
     agent.
- Pass the following explicit context to the agent for each invocation:
  1. The exact path to the current R test file.
  2. The path to the source R library folder.
  3. The path to the corresponding Python library folder (e.g., the root of the `
    r2py_rpart` package, ensuring the agent can correctly locate or create the `tests/`
     directory as per its instructions).
  4. The output folder if specified.

### Step 3: Error Handling and Logging
````

```
- Monitor the output of the ‘@convert-r-test-to-python‘ agent.
- If the agent fails, hallucinates, or throws an error for a specific file, output a
    clear error message to the console noting the specific file that failed.
- **Do not halt** the batch execution. Catch the exception and proceed immediately to
    the next file in the list until all files have been attempted.
```

M.2 Agent: `@convert-r-test-to-python`

```
---
name: convert-r-test-to-python
description: Converts a specified R test file to its Python equivalent using provided
    library contexts.
---

# Convert an R Test File to Python

## Description

Translate a provided R test script into an equivalent Python test script. To ensure
    accurate translation, you must rely on the provided paths to the target R test file
    , the source R library folder, and the corresponding Python library folder. You
    must maintain the exact test coverage, logic, and assertions of the original R code
     while adapting to standard Python testing conventions (e.g., using ‘pytest‘) if
    necessary.

## Execution Steps

### Step 1: Analyze Dependencies and Context

- Read the provided R library directory to understand the function signatures, data
    structures, and logic being tested in the R script.
- Read the provided Python library directory to identify the equivalent Python
    functions, classes, and modules that the new test file will need to import.

### Step 2: Resolve Datasets and Fixtures

- Scan the R test file for any required datasets or external files.
- Look for these files locally first. If they require downloading from an external
    source, prompt the user for the correct URL or source if it is not explicitly
    defined in the R script.
- Ensure the translated Python code uses appropriate libraries (e.g., ‘pandas‘ or built
    -in ‘csv‘) to load these datasets correctly.

### Step 3: Translate the Test Logic

- Translate the R test file into Python syntax. Do not change any logic in the file;
    the workflow should remain exactly identical.
- If the R test file is simply a collection of calls to the R library functions and
    print statements, you must use ‘rpy2‘ to interface with the R environment, call the
     functions in the R library, and replicate the same logic with the Python library,
    and then use ‘assert‘ statements from ‘pytest‘ or ‘unittest‘ to validate the
    alignment of results.
```

```
- If the R test file contains plots or visualizations that are not part of the R
    library, you should ignore them and check the alignment of numerical results
    instead.
- Map R testing framework functions (like those from `testthat`) to their exact Python
    equivalents (e.g., standard `assert` statements for `pytest` or `unittest.TestCase`
     methods), if applicable.
- Pay strict attention to handling 1-based indexing in R versus 0-based indexing in
    Python, as well as differences in how NA/NaN values are treated.

### Step 4: Save the Python Test File

- Check if the user specified an output folder in their prompt.
- If an output folder is specified, save the translated Python test file there.
- If no output folder is specified, create a `tests/` directory inside the provided
    Python library folder (if it doesn't already exist) and save the file there. The
    file should be named identically to the original R test file but with a `.py`
    extension (e.g., `example.R` becomes `example.py`, but if using `pytest`, the file
    would be named `test_example.py`).
```

N Phase 12.1: Public-Interface Test Generation

N.1 Skill: /generate-python-file-tests

```
---
name: generate-python-file-tests
description: Generates Python unit tests for all public interfaces in a given Python
    file by parsing the file, referencing R documentation, and invoking the function
    test generator agent sequentially.
---

# Generate Python File Tests

## Description

When provided with a target Python file path and its corresponding R package
    documentation reference, your task is to parse the file for all function
    definitions, filter out internal helpers to isolate public interfaces, and
    sequentially invoke the `@generate-python-function-tests` agent to build
    comprehensive `pytest` suites for each public function.

## Execution Steps

### Step 1: Extract All Python Function Definitions

* Parse the target Python source file to identify and extract all function definitions.
* Compile a comprehensive list of all function names present in the file, regardless of
     their naming conventions (e.g., ignoring leading underscores for now).

### Step 2: Filter Public Interfaces via R Documentation

* Navigate to the documentation of the original R package (e.g., via CRAN, local `.Rd`
    files, or the package's reference manual). *Example:* For the `rpart` package,
```

```
    review its documentation at ‘https://cran.r-project.org/web/packages/rpart/refman/
    rpart.html‘. Use this to identify all the public interfaces of the R package.
* Cross-reference the comprehensive list of Python functions from Step 1 against the
    exported (public) functions detailed in the R documentation.
* Filter the list to retain *only* the functions that are documented as public
    interfaces in the original R package. Discard internal or private helper functions
    from the execution queue.

### Step 3: Execute Sequential Test Generation

Iterate through the filtered list of public functions from Step 2 sequentially. For
    each function:
1. **Invoke Test Generation Agent:** Call the ‘@generate-python-function-tests‘ agent.
    Provide the following context:
   * The target Python function name.
   * The target Python file path.
   * The original R package/function reference for functional equivalence benchmarking.
2. **Error Handling:** If the agent fails, times out, or throws an error while
    generating tests for a specific function, log a clear error message to the console.
     **Do not halt** the overall batch execution; proceed immediately to the next
    function in the queue.
```

N.2 Agent: @generate-python-function-tests

```
---
name: generate-python-function-tests
description: Generates Python function tests for a given function in a given Python
    file, benchmarking against its original R implementation.
---

# Generate Python Function Tests

## Description

When provided with a target Python function name, a Python file path, and the original
    R function/package reference, your task is to generate comprehensive unit tests
    using ‘pytest‘. The tests must cover positive cases, negative cases, and boundary/
    edge cases to ensure the Python function’s behavior strictly mirrors the original R
     implementation.

## Expected Inputs
- ‘TARGET_FUNCTION‘: The name of the Python function to test.
- ‘FILE_PATH‘: The path to the Python file containing the function.
- ‘R_FUNCTION‘ / ‘R_PACKAGE‘: The original R function and package to benchmark against.

## Execution Steps

### Step 1: Review the Target Function and Its Context

- Navigate to ‘FILE_PATH‘ and locate ‘TARGET_FUNCTION‘.
- Read the function’s implementation, analyzing its parameters, return values, and any
    internal logic or dependencies on other modules.
```

```
- Analyze the docstring to understand the intended behavior, expected inputs, and
    outputs. If there is no docstring, infer the function's purpose from the code.
- Identify specific edge cases or mathematical constraints based on the logic (e.g.,
    division by zero, null checks).

### Step 2: Review the Documentation of the Original Function

- Navigate to the documentation of the original R function (e.g., searching CRAN
    documentation or using R's internal `help()` documentation).
- Extract key information about the function's expected behavior, input parameters,
    default arguments, return values, and any relevant side effects or constraints.
- *Example:* For a function from the `rpart` package, review its documentation at `
    https://cran.r-project.org/web/packages/rpart/refman/rpart.html`. Use this to
    understand the data types it accepts and exactly what output structures it returns.

### Step 3: Generate Positive Tests

Generate at least 8 positive test cases covering all valid input scenarios (typical
    cases and varied data types) using `pytest`. You have to cover all the use cases
    and input types that the original R function supports.
- **All parameters and their combinations must be tested.** To achieve this, you have
    to first extract all the parameters of the Python function and their default values
     (if any) from the docstring or code. Then, you have to generate test cases that
    cover all possible combinations of these parameters.
- **Naming Convention:** Save the tests in the Python project's `tests/` directory as `
    test_<TARGET_FUNCTION>_positive.py`. (Create the `tests/` directory if it does not
    exist). If the file already exists, append the new tests to it without removing any
     existing tests. It is fine if you find existing tests are enough to cover all
    scenarios, in that case, you can skip the generation of additional tests.
- **Setup:** Define descriptive test functions (e.g., `test_<TARGET_FUNCTION>
    _with_standard_array`). Set up necessary input data mimicking valid R data
    structures.
- **R Execution:** Use `rpy2` to execute the original R function with the test input
    data and retrieve the expected output.
- **Python Execution & Assertion:** Call the target Python function with the identical
    input data.
- **Comparison:** Assert that the outputs match. *Crucial:* When comparing floats or
    numerical arrays between R and Python, use `numpy.testing.assert_allclose` or `math.
    isclose` to account for minor floating-point precision differences between the two
    languages.

### Step 4: Generate Negative Tests

Generate negative test cases covering invalid input scenarios to ensure the Python
    function handles errors identically to the R function. Scan the Python function and
     all related functions for all error-handling logic (e.g., `raise ValueError`); all
     branches must be tested.
- **Naming Convention:** Save as `test_<TARGET_FUNCTION>_negative.py` in the `tests/`
    directory in the Python project. If the file already exists, append the new tests
    to it without removing any existing tests. It is fine if you find existing tests
    are enough to cover all scenarios, in that case, you can skip the generation of
    additional tests.
- **Setup:** Define inputs that should trigger failures (e.g., incorrect data types,
    out-of-bounds parameters, missing required arguments).
```

```
- **Execution & Comparison:** 1. Use `rpy2` to call the R function and catch the
    resulting R error message as a string.
  2. Use `pytest.raises(Exception) as exc_info` to catch the Python function's error.
  3. **Pass Condition:** If both functions raise an error, the test passes. If one
    raises an error and the other does not, the test fails.
  4. **Warning Condition:** Compare the text of the Python error message (`str(exc_info
    .value)`) with the caught R error message. If they differ in phrasing, do not fail
    the test, but use Python's `warnings.warn()` to print a warning indicating the
    discrepancy in error messaging.

### Step 5: Generate Boundary and Edge Case Tests

Generate at least 8 tests focusing strictly on the functional limits and extremes. You
    have to cover all edge cases that the original R function handles, such as minimum/
    maximum values, `NaN`, `Inf`, empty inputs, and singleton values. The goal is to
    ensure that the Python function's behavior in these edge cases is identical to the
    R function's behavior.
- **Naming Convention:** Save as `test_<TARGET_FUNCTION>_edge.py` in the `tests/`
    directory in the Python project. If the file already exists, append the new tests
    to it without removing any existing tests. It is fine if you find existing tests
    are enough to cover all scenarios, in that case, you can skip the generation of
    additional tests.
- **Setup:** Use inputs such as minimum/maximum possible values, `NaN`, `Inf`, empty
    strings, empty arrays/DataFrames, or singleton values.
- **Execution:** Determine whether the R function resolves these extremes with a valid
    output or an error.
- **Comparison:** Use the exact same validation logic applied in Step 3 (for successful
     outputs) or Step 4 (for expected errors) to ensure the Python script's edge-case
    handling is identical to R's.
```

O Phase 12.2: Bug Resolution

O.1 Skill: `/fix-bugs-found-in-tests`

```
---
name: fix-bugs-found-in-tests
description: Automatically runs a test suite, identifies failing tests, invokes the fix
    -bug-found-in-test agent to resolve them, and repeats until all tests pass.
---

# Fix Bugs Found in Tests

## Description

You will be provided with a folder containing Python test files, a target Python
    library folder, and the corresponding original R library folder. Your task is to
    execute the entire test suite, identify any test failures, and systematically
    invoke the `@fix-bug-found-in-test` agent to resolve the root cause of each failure.
     You will repeat this test-and-invoke cycle autonomously until the entire test
    suite passes.

## Execution Steps
```

```
### Step 1: Execute the Test Suite

- Navigate to the designated folder containing the Python test files.
- Run the full test suite using ‘pytest‘ for all files matching the pattern ‘test_*.py
    ‘.
- Capture the full terminal output from ‘pytest‘. If all tests pass, exit successfully.
- If there are failures, isolate the very first failing test to begin the debugging
    process. Address failures sequentially.

### Step 2: Invoke the Fix Agent

- For the isolated failing test, invoke the ‘@fix-bug-found-in-test‘ agent.
- Ensure you pass the agent the necessary context it requires: the specific failing
    Python test (the output error message, the function, and the test file), the Python
     library folder, and the corresponding original R library folder.
- Allow the ‘@fix-bug-found-in-test‘ agent to complete its execution.

### Step 3: Validate and Loop

- Once the ‘@fix-bug-found-in-test‘ agent has finished its task, reinstall the fixed
    package, and then rerun the entire test suite using ‘pytest‘ across all ‘test_*.py‘
     files to ensure the fix worked and no regressions were introduced.
- If new or remaining tests fail, repeat Steps 1 through 3 for the next failing test.
- Continue this cycle until ‘pytest‘ returns a 100% pass rate.
- *Safety Constraint:* If the ‘@fix-bug-found-in-test‘ agent attempts to fix the exact
    same failing test 3 times consecutively without the test passing, halt the loop and
     request user intervention to prevent an infinite loop.
```

O.2 Agent: `@fix-bug-found-in-test`

```
---
name: fix-bug-found-in-test
description: Analyzes and fixes a failing test for a Python function to ensure it
    correctly reflects the corresponding R library's behavior.
---

# Fix Bug Found in a Test

## Description

You will be provided with a failing Python test, a Python library folder, and the
    corresponding original R library folder. The test is currently failing due to an
    assertion error.

Your task is to analyze the test, compare the Python implementation to the original R
    implementation, and fix the root cause of the failure. The ultimate goal is to
    ensure the Python test passes successfully and accurately validates that the Python
     function matches the R function's behavior.

## Execution Steps

### Step 1: Analyze the Failing Test
```

```
- **Review the Test:** Understand the test function's objective and the expected
    behavior of the Python function being evaluated.
- **Analyze the Failure:** Identify the specific failing assertion. Review the error
    message to understand the discrepancy between the expected and actual outputs.
- **Validate Test Setup:** Check if the test inputs, mock data, and parameters are
    correctly structured and properly passed to the function.

### Step 2: Compare with the Original R Implementation

- **Review R Logic:** Examine the original R implementation of the target function to
    understand its expected outputs for the given inputs.
- **Compare Implementations:** Cross-reference the logic, data handling, and output
    formatting of the R function against the Python function.
- **Identify Edge Cases:** Note any specific scenarios or edge cases handled by the R
    function that might be missing or misrepresented in the Python implementation or
    the test itself.

### Step 3: Implement the Fix

- **Prioritize Consistency:** Your primary directive is to ensure logical consistency
    between the Python library and the original R library.
- **Determine the Root Cause:** Identify whether the assertion failure is due to a
    flawed test (e.g., incorrect expected output, bad inputs) or a bug in the Python
    library code itself.
- **Apply the Correction:** - If the **test** is flawed: Modify the test inputs,
    expected outputs, or assertion structure so it correctly expects the R
    implementation's behavior.
  - If the **Python library** is flawed: Update the Python library code to match the R
    implementation so the test passes.
- **Document Intentional Deviations:** If fixing the bug requires introducing a
    deliberate logical inconsistency between the Python and R libraries, you must
    document this clearly in the code comments and provide a rationale for the
    divergence.
- **Verify:** Ensure the test now runs and passes successfully.
```