Package 'saber'

November 10, 2026

Type Package

Title Context Engineering for Large Language Model Agents

Version 0.8.0

Date 2026-11-09

Description Context-engineering primitives for Artificial Intelligence (AI) coding agents working in R. Assembles agent context from memory and instruction files ('AGENTS.md', 'CLAUDE.md'), traces function call blast radius across projects, generates project briefings, parses source into Abstract Syntax Tree (AST) symbol indices, discovers dependency graphs, and introspects installed packages. Zero dependencies.

License Apache License (>= 2)

Depends R (>= 4.4.0)

URL https://github.com/cornball-ai/saber

BugReports https://github.com/cornball-ai/saber/issues

Suggests bonsaisitter, tinytest

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Troy Hernandez [aut, cre] (ORCID: <https://orcid.org/0009-0005-4248-604X>), cornball.ai [cph]

Maintainer Troy Hernandez <troy@cornball.ai>

Repository CRAN

Date/Publication 2026-11-10 06:20:02 UTC

Contents

agent_context 2
blast_radius 4

briefing 5
context_audit 6
context_manifest 7
context_render 8
default_exclude 9
default_src_exclude 10
find_downstream 10
fn_graph 11
git_commit_count_since 12
git_log_since 13
graph_svg 14
heartbeat 15
pkg_exports 16
pkg_graph 16
pkg_help 17
pkg_internals 18
projects 18
skill_manifest 19
skill_read 20
src_symbols 21
symbols 22

Index **24**

`agent_context` *Load context files for an AI coding agent*

Description

Returns assembled context (memory, project/global instructions, agent identity files) tailored to a specific consumer. Files autoloaded by the agent natively are skipped to avoid duplication.

Usage

```
agent_context(
  agent = NULL,
  project_dir = getwd(),
  workspace_dir = NULL,
  memory_base = file.path(path.expand("~"), ".claude", "projects"),
  claude_global_path = file.path(path.expand("~"), ".claude", "CLAUDE.md"),
  include_memory = NULL,
  include_project = NULL,
  include_global = NULL,
  include_soul = NULL,
  max_memory_lines = 100L
)
```

Arguments

agent	Consumer identifier: "claude", "codex", "corteza", or NULL (interactive / unknown). The legacy "llamar" identifier is accepted as an alias for "corteza".
project_dir	Project directory to scan for CLAUDE.md / AGENTS.md.
workspace_dir	Optional directory containing SOUL.md and USER.md (e.g. ~/.corteza/workspace). If NULL, those files are skipped.
memory_base	Base directory for 'Claude Code' project memory files.
claude_global_path	Path to the global 'Claude Code' instructions file. Defaults to ~/.claude/CLAUDE.md.
include_memory	Override default for memory inclusion. Use TRUE/FALSE to force, or NULL for the agent default.
include_project	Override default for project instructions (CLAUDE.md / AGENTS.md).
include_global	Override default for global instructions (~/.claude/CLAUDE.md / USER.md).
include_soul	Override default for SOUL.md inclusion.
max_memory_lines	Maximum lines to include from each memory source.

Details

Defaults per agent:

- "claude" - skips Claude Code project memory and CLAUDE.md files (autoloaded by 'Claude Code'). Loads Codex memories, AGENTS.md, USER.md, and SOUL.md when present.
- "codex" - skips AGENTS.md and Codex memories (autoloaded by Codex). Loads Claude Code project memory, CLAUDE.md, USER.md, and SOUL.md when present.
- "corteza" or NULL - loads everything available.

Project instructions merge CLAUDE.md and AGENTS.md. The file the consumer autoloads is left out, and so is any heading-delimited section whose text the consumer already holds. Two names for one file, or two identical copies, add nothing; a consumer with no autoloaded file receives CLAUDE.md followed by the sections only AGENTS.md has. Sections are compared verbatim, so the same rule in different words is loaded from both files.

Global instructions are resolved by trying both naming conventions and picking the file relevant to the consumer: ~/.claude/CLAUDE.md or <workspace_dir>/USER.md.

Override the defaults with the include_* parameters.

Value

Character string of assembled context, or empty string if no context applies.

Examples

```
# Codex agent in current project
saber::agent_context(agent = "codex")

# Corteza with workspace files
saber::agent_context(agent = "corteza",
                     workspace_dir = "~/.corteza/workspace")

# Force-include memory regardless of agent default
saber::agent_context(agent = "claude", include_memory = TRUE)
```

---

blast_radius *Find callers of a function across projects*

Description

Given a function name and project, finds all internal callers within that project and all callers in downstream projects (projects whose DESCRIPTION lists this one in Depends, Imports, or LinkingTo).

Usage

```
blast_radius(
  fn,
  project = NULL,
  include = "r",
  scan_dir = path.expand("~"),
  cache_dir = file.path(tools::R_user_dir("saber", "cache"), "symbols"),
  exclude = default_exclude()
)
```

Arguments

fn	Character. Function name to search for.
project	Character. Project name (or path to project directory).
include	Character vector. Any of "r" (R source, default), "examples" (roxygen @examples blocks in the target project), "vignettes" (code chunks in the target project's vignettes), and "src" (the target project's C, C++, Python, Rust, and JavaScript sources).
scan_dir	Directory to scan for downstream projects.
cache_dir	Directory for symbol cache files.
exclude	Character vector of directory basenames to skip when scanning for downstream projects.

Details

With `include = c("r", "examples", "vignettes")` the search can be extended to references in the target project's roxygen `@examples` blocks and vignette code chunks (Rmd, qmd, Rnw). Documentation scanning is target-project only; it does not walk downstream projects' docs.

With `include = "src"` the target project's C, C++, Python, Rust, and JavaScript sources are searched via `src_symbols`, reporting their callers of `fn`. This requires the suggested **bonsaisitter** runtime plus a grammar package per language (see `src_symbols`) and, like documentation scanning, is target-project only.

Value

A data.frame with columns: caller, project, file, line, source. `source` is one of `"r"`, `"example"`, `"vignette"`, `"src"`.

Examples

```
# Create a minimal project
d <- file.path(tempdir(), "blastpkg")
dir.create(file.path(d, "R"), recursive = TRUE, showWarnings = FALSE)
writeLines("helper <- function(x) x + 1", file.path(d, "R", "helper.R"))
writeLines("main <- function(x) helper(x * 2)", file.path(d, "R", "main.R"))

# Find all callers of helper()
blast_radius("helper", project = d, scan_dir = tempdir(),
             cache_dir = tempdir())

# Include roxygen @examples and vignettes from the target project
blast_radius("helper", project = d, include = c("r", "examples", "vignettes"),
             scan_dir = tempdir(), cache_dir = tempdir())
```

`briefing` *Generate a project briefing*

Description

Produces a concise markdown briefing combining DESCRIPTION metadata, downstream dependents, and recent git commits. Written to the user cache directory so both the agent and user see the same context.

Usage

```
briefing(
  project = NULL,
  scan_dir = path.expand("~"),
  briefs_dir = file.path(tools::R_user_dir("saber", "cache"), "briefs")
)
```

Arguments

project	Project name. If NULL, inferred from the current working directory basename.
scan_dir	Directory to scan for project directories.
briefs_dir	Directory to write briefing markdown files.

Details

For runtime context (memory, identity files, project instructions), see agent_context.

Value

The briefing text (character string), returned invisibly. Emitted via message() and written to briefs_dir/{project}.md.

Examples

```
d <- file.path(tempdir(), "briefpkg")
dir.create(file.path(d, "R"), recursive = TRUE, showWarnings = FALSE)
writeLines(c("Package: briefpkg", "Title: Demo", "Version: 0.1.0"),
           file.path(d, "DESCRIPTION"))
briefing("briefpkg", scan_dir = tempdir(),
         briefs_dir = file.path(tempdir(), "briefs"))
```

---

context_audit *Audit context inclusion and token costs*

Description

Reports metadata, exact duplicate sources, native overlap, missing or unreadable files, audience exclusions, explicit truncation, and size thresholds. Does not print or retain source bodies in its result, execute snippets, or infer semantic contradictions from arbitrary text.

Usage

```
context_audit(manifest, layer_tokens = 2000L, total_tokens = 8000L)
```

Arguments

manifest	A manifest returned by context_manifest.
layer_tokens	Estimated-token warning threshold for one source.
total_tokens	Estimated-token warning threshold for rendered context.

Value

A saber_context_audit list with metadata-only sources, findings (source_id, code, severity, related_id), total emitted sizes, and the estimator label. Native tokens are not part of the emitted total.

Examples

```
m <- context_manifest("corteza", discover = FALSE,
    extra_sources = list(list(id = "note", kind = "memory", text = "A note.")))
context_audit(m, layer_tokens = 1)
```

`context_manifest` *Build a source manifest for agent context*

Description

Discovers shared and project instructions, records source provenance, and selects context without changing the legacy `agent_context` API. Printing the result shows metadata only. Source text is available through `context_render` and the manifest's `fragments` field.

Usage

```
context_manifest(
  agent,
  project_dir = getwd(),
  workspace_dir = NULL,
  shared_path = NULL,
  native_paths = character(),
  extra_sources = list(),
  budgets = list(),
  discover = TRUE
)
```

Arguments

`agent`	Nonempty consumer name, such as `"corteza"`.
`project_dir`	Project directory. Relative source paths resolve here.
`workspace_dir`	Optional directory containing USER.md and SOUL.md.
`shared_path`	Shared instruction file. NULL uses AGENTS_GLOBAL_MD, then ~/.config/agents/GLOBAL.md. FALSE disables shared-file discovery.
`native_paths`	Character vector of files the caller confirms are already loaded by the consumer. These are measured but never reinjected.
`extra_sources`	List of source descriptors, described below.
`budgets`	Named list of budgets keyed by source id or kind. Each contains max_chars and/or max_lines, nonnegative integers or Inf. A source-id budget takes precedence over a kind budget. Kind budgets are shared by matching sources in rendering order.
`discover`	Include standard shared, project, and workspace sources? FALSE uses only native_paths and extra_sources.

Details

Discovery prefers AGENTS.md and falls back to CLAUDE.md when AGENTS.md is missing, empty, or unreadable. Both candidates remain in the manifest. Shared preferences and workspace USER.md are independent sources. The Claude global file, memories, ancestor instructions, skills, and briefings are not discovered implicitly. Supply those as explicit sources. Native loading is declared by the caller, never inferred from agent alone.

Each extra source is a list with a unique `id`, a `kind`, and exactly one of `path` or scalar UTF-8 `text`. Optional fields are `audience` (character vector, default `"*"`), `delivery` (`"consumer"`, `"hook"`, or `"native"`), `origin`, `scope`, `config_path`, `native_evidence`, and numeric `order`. Defaults preserve discovery then extra-source order; set order to place generated layers before or between discovered sources.

Files are read as UTF-8 without rewriting whitespace or line endings. Deduplication compares canonical paths and exact bytes before budgets, with native sources for the current audience taking precedence regardless of order. Missing and unreadable native files can still suppress the same path, but cannot establish equality to other files. MD5 fingerprints identify content, not authenticity; byte equality confirms content matches.

Budgets apply to emitted fragments, excluding the blank-line separators inserted by context_render(). Native sources do not consume these budgets. Truncation is recorded, with no unbudgeted marker inserted into the text. By default nothing is truncated. Lines count physical lines, excluding an empty line after a final newline; tokens are ceiling(characters / 4). No source code is evaluated and no persistent cache is written.

Value

A `saber_context_manifest` list with schema_version, agent, project_dir, estimator, budgets, a sources data frame, and named fragments. The data frame records paths, audience, delivery, inclusion reasons, duplicate ids, source and emitted sizes/hashes, and budget loss. `requested_path` preserves the original source path, `path` resolves it against project_dir and expands tilde, and `canonical_path` resolves existing paths and symbolic links. Generated text uses empty paths.

Examples

```
m <- context_manifest("corteza", discover = FALSE,
    extra_sources = list(list(id = "runtime", kind = "runtime",
                              text = "R objects persist across turns.")))
context_audit(m)
cat(context_render(m))
```

`context_render` *Render the selected context fragments*

Description

Returns included injectable fragments in manifest order, joined with two newlines. Native and skipped sources are excluded. This is an explicit request for source contents; printing a manifest or audit does not show them.

Usage

```
context_render(manifest)
```

Arguments

manifest	A manifest returned by context_manifest.

Value

A character string, or "" when no fragments are included.

Examples

```
m <- context_manifest("corteza", discover = FALSE,
    extra_sources = list(list(id = "note", kind = "memory", text = "A note.")))
context_render(m)
```

---

default_exclude *Default directories to exclude when scanning for projects*

Description

Returns a character vector of directory basenames that are skipped when scanning for downstream projects. Override by passing a custom exclude vector to blast_radius.

Usage

```
default_exclude()
```

Value

Character vector of directory basenames.

Examples

```
default_exclude()
```

default_src_exclude *Default directories to exclude when scanning for source files*

Description

Returns a character vector of directory basenames that `src_symbols` skips while scanning: everything in `default_exclude` (user directories such as `Documents`, plus caches and build artifacts), extended with dependency and build trees whose sources are not the project's own (including cargo's `target`). Hidden directories (e.g. `.git`) and `*.Rcheck` directories are always skipped, whatever the `exclude` value. Extend it for vendored code, e.g. `c(default_src_exclude(), "tree-sitter")`.

Usage

```
default_src_exclude()
```

Value

Character vector of directory basenames.

Examples

```
default_src_exclude()
```

find_downstream *Find projects that depend on a given package*

Description

Scans DESCRIPTION files in project directories under `scan_dir` for Depends, Imports, or LinkingTo fields that reference `package`.

Usage

```
find_downstream(
  package,
  scan_dir = path.expand("~"),
  exclude = default_exclude()
)
```

Arguments

`package`	Character. Package name to search for.
`scan_dir`	Directory to scan for project directories.
`exclude`	Character vector of directory basenames to skip.

Value

Character vector of project names that depend on package.

Examples

```
d <- file.path(tempdir(), "dsdir")
dir.create(d, showWarnings = FALSE)
pkg <- file.path(d, "child")
dir.create(pkg, showWarnings = FALSE)
writeLines(c("Package: child", "Version: 0.1.0", "Imports: parent"),
           file.path(pkg, "DESCRIPTION"))
find_downstream("parent", scan_dir = d)
```

---

fn_graph *Render a function call graph for an R project*

Description

Pulls the AST symbol index via `symbols` and renders internal function-to-function call edges as SVG. External calls (into other packages) are dropped by default.

Usage

```
fn_graph(
  project_dir,
  include_external = FALSE,
  ...,
  cache_dir = file.path(tools::R_user_dir("saber", "cache"), "symbols")
)
```

Arguments

`project_dir`	Path to the project directory (an R package root).
`include_external`	If `TRUE`, also include nodes for functions called from other packages. Default `FALSE`.
`...`	Passed through to `graph_svg` (e.g., `width`, `height`, `iterations`, `seed`).
`cache_dir`	Directory for the underlying `symbols` cache. Pass `tempdir()` (or any non-default path) when running examples / tests to avoid writing to the user's persistent cache. Name-only argument; pass it by name.

Value

Character vector of SVG lines. Write with `writeLines()`.

Examples

```
d <- file.path(tempdir(), "fngdemo")
dir.create(file.path(d, "R"), recursive = TRUE, showWarnings = FALSE)
writeLines(c("Package: demo", "Version: 0.1.0"),
           file.path(d, "DESCRIPTION"))
writeLines("add <- function(x, y) x + y", file.path(d, "R", "add.R"))
writeLines("double <- function(x) add(x, x)",
           file.path(d, "R", "double.R"))
svg <- fn_graph(d, cache_dir = tempdir())
writeLines(svg, tempfile(fileext = ".svg"))
```

`git_commit_count_since` *Count commits in a repository since a date*

Description

Returns the number of commits on `HEAD` dated on or after `since_date`, and `0L` for anything that is not a git working tree (plain directories, missing paths). Worktrees and dubious-ownership repositories are handled: a `git rev-parse` check runs first, and git's non-zero exits never leak as warnings. The counting primitive behind `heartbeat`, exported for downstream tooling that composes its own activity reports.

Usage

```
git_commit_count_since(repo_dir, since_date)
```

Arguments

`repo_dir`	Path to a repository.
`since_date`	A `Date` (or string in `YYYY-MM-DD` form). Commits from 00:00 local time on this date count.

Value

Integer commit count.

Examples

```
# A plain directory is not a repository: counts as zero
git_commit_count_since(tempdir(), Sys.Date() - 7)
```

`git_log_since` *List recent commits in a repository*

Description

Returns up to `n` commit lines from `HEAD`, newest first. With `format = "oneline"` each line is git's `--oneline` (short hash and subject); with `format = "iso"` each line is tab-separated ISO-local timestamp, short hash, and subject. A `NULL` `since_date` means no lower bound. Anything that is not a git working tree returns `character(0)`. The log primitive behind `heartbeat`, exported for downstream tooling that composes its own activity reports.

Usage

```
git_log_since(
  repo_dir,
  since_date = NULL,
  n = 5L,
  format = c("oneline", "iso")
)
```

Arguments

`repo_dir`	Path to a repository.
`since_date`	A `Date` (or string in `YYYY-MM-DD` form), or `NULL` for no lower bound. Commits from 00:00 local time on this date are included.
`n`	Integer. Maximum commits to return.
`format`	`"oneline"` (default) or `"iso"`.

Value

Character vector of commit lines (possibly empty).

Examples

```
# A plain directory is not a repository: empty log
git_log_since(tempdir(), Sys.Date() - 7)
```

graph_svg *Render a graph as static SVG*

Description

Runs a Fruchterman-Reingold force simulation to lay out nodes, then emits SVG with baked coordinates. Output is interactive via native browser features: hover tooltips from `<title>` elements, click navigation from `<a xlink:href>` wrappers, CSS `:hover` highlighting. No JavaScript.

Usage

```
graph_svg(
  edges,
  nodes = NULL,
  width = 1200L,
  height = 900L,
  iterations = 50L,
  seed = 1L
)
```

Arguments

edges	Data frame with `from` and `to` columns holding node ids.
nodes	Optional data frame with `id`, `label`, `href` columns. Optionally a `tooltip` column (plain text, may contain newlines) that overrides the default hover text (which is the label). `id` must cover every node mentioned in `edges`. If `NULL`, ids from `edges` are used as labels with no hrefs or tooltips.
width	SVG viewport width in pixels.
height	SVG viewport height in pixels.
iterations	Force-simulation steps. 50 is usually enough.
seed	Integer seed for the random initial layout (output is deterministic given the same seed).

Details

Suitable for graphs up to roughly a few hundred nodes. The repulsion step is vectorized via `outer()` but allocates an `n` x `n` matrix per iteration; larger graphs should pre-filter.

Value

Character vector, one SVG element per line. Write with `writeLines()`.

Examples

```
edges <- data.frame(from = c("a", "a", "b"),
                    to = c("b", "c", "c"))
svg <- graph_svg(edges)
writeLines(svg, tempfile(fileext = ".svg"))
```

`heartbeat` *Summarize recent git activity across projects*

Description

Scans every git repository directly under `scan_dir`, counts commits inside the lookback window, and produces a markdown summary of the active projects with their most recent commits, busiest first. The complement to `briefing`: `briefing()` is one project in depth, `heartbeat()` is every project at a glance.

Usage

```
heartbeat(
  days = 7L,
  scan_dir = path.expand("~"),
  n = 5L,
  exclude = default_exclude(),
  briefs_dir = file.path(tools::R_user_dir("saber", "cache"), "briefs")
)
```

Arguments

`days`	Integer. Lookback window in days.
`scan_dir`	Directory whose immediate subdirectories are scanned.
`n`	Integer. Maximum commits to list per active project.
`exclude`	Character vector of directory basenames to skip.
`briefs_dir`	Directory to write the heartbeat markdown file.

Value

The heartbeat text (character string), returned invisibly. Emitted via `message()` and written to `briefs_dir/_heartbeat.md`.

Examples

```
root <- file.path(tempdir(), "hbscan")
dir.create(root, showWarnings = FALSE)
heartbeat(days = 7, scan_dir = root,
          briefs_dir = file.path(tempdir(), "briefs"))
```

pkg_exports *List exported functions of a package*

Description

Returns a data.frame of exported functions with their argument signatures.

Usage

```
pkg_exports(package, pattern = NULL)
```

Arguments

package	Character. Package name.
pattern	Optional regex to filter function names.

Value

A data.frame with columns: name, args.

Examples

```
pkg_exports("tools")
pkg_exports("tools", pattern = "^Rd")
```

pkg_graph *Render a package-level dependency graph*

Description

Discovers R packages under `scan_dir` via `projects`, parses each one's `Depends` and `Imports` fields, and renders edges between packages that both live in `scan_dir`. External CRAN dependencies are dropped.

Usage

```
pkg_graph(
  scan_dir = path.expand("~"),
  packages = NULL,
  include_suggests = FALSE,
  ...
)
```

Arguments

scan_dir	Directory to scan for project directories.
packages	Optional character vector limiting the graph to these packages.
include_suggests	If TRUE, also include edges for packages in each project's Suggests field. Default FALSE (only Depends and Imports).
...	Passed through to graph_svg.

Value

Character vector of SVG lines. Write with writeLines().

Examples

```
d <- file.path(tempdir(), "pkgdemo")
dir.create(file.path(d, "parent"), recursive = TRUE, showWarnings = FALSE)
dir.create(file.path(d, "child"), showWarnings = FALSE)
writeLines(c("Package: parent", "Title: P", "Version: 0.1.0"),
           file.path(d, "parent", "DESCRIPTION"))
writeLines(c("Package: child", "Title: C", "Version: 0.1.0",
             "Imports: parent"),
           file.path(d, "child", "DESCRIPTION"))
svg <- pkg_graph(scan_dir = d)
writeLines(svg, tempfile(fileext = ".svg"))
```

---

pkg_help *Get help for a package topic as markdown*

Description

Extracts help documentation and converts it to clean markdown.

Usage

```
pkg_help(topic, package, format = c("md", "hugo"))
```

Arguments

topic	Character. The help topic name.
package	Character. Package name.
format	Character. Output format: "md" (default) for plain markdown, or "hugo" for Hugo-compatible markdown with YAML front matter.

Value

Character string of markdown help text.

Examples

```
cat(pkg_help("md5sum", "tools"))
```

`pkg_internals` *List internal (non-exported) functions of a package*

Description

Returns functions defined in a package namespace but not exported.

Usage

```
pkg_internals(package, pattern = NULL)
```

Arguments

`package`	Character. Package name.
`pattern`	Optional regex to filter function names.

Value

A data.frame with columns: name, args.

Examples

```
pkg_internals("tools", pattern = "^check")
```

`projects` *Discover R package projects*

Description

Scans a directory for subdirectories containing a DESCRIPTION file and returns their metadata. Symbolic-link aliases of the same directory are reported once, preferring its non-alias path when present. Separate checkouts remain separate even when they declare the same package name.

Usage

```
projects(scan_dir = path.expand("~"), exclude = default_exclude())
```

Arguments

`scan_dir`	Directory to scan for project directories.
`exclude`	Character vector of directory basenames to skip.

Value

A data.frame with columns: package, title, version, path, depends, imports.

Examples

```
d <- file.path(tempdir(), "scandir")
dir.create(d, showWarnings = FALSE)
pkg <- file.path(d, "mypkg")
dir.create(pkg, showWarnings = FALSE)
writeLines(c("Package: mypkg", "Title: Demo", "Version: 0.1.0"),
           file.path(pkg, "DESCRIPTION"))
projects(scan_dir = d)
```

---

skill_manifest *Discover package and personal instruction skills*

Description

Builds a read-only inventory without loading packages, executing skills, changing registrations, or writing a cache. Flat package bundles and the older package-nested layout are both supported.

Usage

```
skill_manifest(
  packages = NULL,
  lib.loc = .libPaths(),
  project_dirs = character(),
  roots = character(),
  prefer = c("source", "installed"),
  disabled = character()
)
```

Arguments

packages	Installed package names to inspect. NULL scans all packages in lib.loc; character() skips installed packages.
lib.loc	R library directories, in precedence order.
project_dirs	Explicit package checkout directories (not scan roots).
roots	Optional named character vector of non-package skill roots.
prefer	Package source precedence, either source or installed first.
disabled	Exact package-qualified or root-qualified skill ids to omit.

Details

Package ids have the form package:NAME/SKILL; configured roots use root:NAME/SKILL. Package selection is whole-root: the preferred origin wins, then the first explicit checkout or library. A partial source bundle does not borrow skills from a different installed version. Ambiguous skill names within the selected root are excluded with diagnostics, not overwritten.

Discovery reads single-line, quoted, and folded/literal name/description frontmatter. It is not a general YAML parser and never evaluates metadata. Supporting files are fingerprinted for later drift checks; MD5 fingerprints detect changes, not authenticity. Hidden files are not instruction resources. Symlinks within a root are followed once; escapes are refused. No assumption is made about a user's home directory or personal skill layout.

Existing agent context APIs and native skill registrations are unchanged. Feed selected metadata to the consumer's catalog and use skill_read() for on-demand instruction retrieval; discovery alone does not register tools.

Value

A saber_skill_manifest with entries, roots, resource fingerprints, and diagnostics. Printing exposes metadata only, never instruction bodies.

Examples

```
m <- skill_manifest(packages = character(), lib.loc = character())
print(m)
```

`skill_read` *Read an instruction from a skill manifest*

Description

Returns the exact UTF-8 file text after checking its recorded identity and fingerprint. Supporting resources receive the same checks as SKILL.md.

Usage

```
skill_read(manifest, id, resource = "SKILL.md")
```

Arguments

`manifest`	A skill_manifest() result for the current session.
`id`	Exact selected skill id, including its package or root namespace.
`resource`	Relative resource path within that skill, default SKILL.md.

Details

Only resources inventoried in this manifest are readable. Absolute paths, parent traversal, changed files, and symlink escapes are refused. Rebuild the manifest explicitly to accept changed source instructions. This reader does not execute scripts or grant tool permissions. Binary/NUL-containing files and files larger than 1 MiB are refused by this text interface.

Value

One character string, without a wrapper or whitespace normalization.

Examples

```
m <- skill_manifest(packages = character(), lib.loc = character())
nrow(m$entries)
```

`src_symbols` *Build a symbol index for a project's non-R sources*

Description

Recursively scans `project_dir` for C (`.c`, `.h`), C++ (`.cc`, `.cpp`, `.cxx`, `.hh`, `.hpp`), Python (`.py`), Rust (`.rs`), and JavaScript (`.js`, `.mjs`, `.cjs`, `.jsx`) sources and parses them into function definitions and call relationships, mirroring the shape of `symbols` with an added `lang` column. This covers the `src/` directory of an R package as well as repositories that are not R packages at all. Directories named in `exclude` are skipped, as are hidden directories and `*.Rcheck` directories, always. Results are cached as RDS in the user cache directory alongside the `symbols()` cache.

Usage

```
src_symbols(
  project_dir,
  langs = c("c", "cpp", "python", "rust", "javascript"),
  exclude = default_src_exclude(),
  cache_dir = file.path(tools::R_user_dir("saber", "cache"), "symbols")
)
```

Arguments

`project_dir`	Path to the project directory.
`langs`	Character vector of languages to index. Any of `"c"`, `"cpp"`, `"python"`, `"rust"`, `"javascript"` (all five by default).
`exclude`	Character vector of directory basenames to skip while scanning, e.g. vendored or generated trees. The default `default_src_exclude` includes the `default_exclude` opt-outs, so user directories such as `Documents` stay untouched even when scanning from a home directory.
`cache_dir`	Directory for symbol cache files.

Details

A definition is marked `exported` when it is visible beyond its own file or module: for C/C++, definitions not declared `static`; for Python, names without a leading underscore; for Rust, items declared `pub`; for JavaScript, definitions wrapped in an `export` statement.

Parsing uses the suggested **bonsaisitter** tree-sitter runtime with one grammar package per language: **treesitter.c** for C (falling back to **treesitter.cpp**, which also parses C), **treesitter.cpp** for C++, **treesitter.python** for Python, **treesitter.rust** for Rust, and **treesitter.javascript** for JavaScript. Grammars are only required for languages that actually match files, and projects with no matching files return empty indices without requiring any of them.

Value

A list with components:

defs data.frame(name, file, line, lang, exported)

calls data.frame(caller, callee, file, line, lang)

Examples

```
# Needs the bonsaisitter runtime; also needs a grammar package per
# language, tolerated via tryCatch() so the example survives its absence
if (requireNamespace("bonsaisitter", quietly = TRUE)) {

    # Create a minimal project with a src/ directory
    d <- file.path(tempdir(), "srcdemo")
    dir.create(file.path(d, "src"), recursive = TRUE, showWarnings = FALSE)
    writeLines(c(
        "static double square(double x) { return x * x; }",
        "double area(double r) { return 3.14159 * square(r); }"
    ), file.path(d, "src", "area.c"))

    idx <- tryCatch(src_symbols(d, cache_dir = tempdir()),
                    error = function(e) NULL)
    idx$defs   # C function definitions
    idx$calls  # call relationships (area calls square)
}
```

`symbols` *Build a symbol index for a project*

Description

Parses all `R/*.R` files in a project directory using `getParseData()` to extract function definitions and function calls. Results are cached as RDS in the user cache directory.

Usage

```
symbols(
  project_dir,
  cache_dir = file.path(tools::R_user_dir("saber", "cache"), "symbols")
)
```

Arguments

project_dir	Path to the project directory.
cache_dir	Directory for symbol cache files.

Value

A list with components:

defs data.frame(name, file, line, exported)

calls data.frame(caller, callee, file, line)

Examples

```
# Create a minimal project with R source files
d <- file.path(tempdir(), "demopkg")
dir.create(file.path(d, "R"), recursive = TRUE, showWarnings = FALSE)
writeLines("add <- function(x, y) x + y", file.path(d, "R", "add.R"))
writeLines("double <- function(x) add(x, x)", file.path(d, "R", "double.R"))

idx <- symbols(d, cache_dir = tempdir())
idx$defs   # function definitions
idx$calls  # call relationships (double calls add)
```

Index

agent_context, 2, *6, 7*

blast_radius, 4, *9*
briefing, 5, *15*

context_audit, 6
context_manifest, *6*, 7, *9*
context_render, *7*, 8

default_exclude, 9, *10*, *21*
default_src_exclude, 10, *21*

find_downstream, 10
fn_graph, 11

git_commit_count_since, 12
git_log_since, 13
graph_svg, *11*, 14, *17*

heartbeat, *12, 13*, 15

pkg_exports, 16
pkg_graph, 16
pkg_help, 17
pkg_internals, 18
projects, *16*, 18

skill_manifest, 19
skill_read, 20
src_symbols, *5*, *10*, 21
symbols, *11*, *21*, 22