Package: gptr (via r-universe)

November 1, 2026

Title Language Model Agents in Live 'R' Sessions

Version 1.0.0

Description Runs language model agents in an existing 'R' session, where generated code can inspect and modify objects already in memory. The peter() function supports interactive chat and programmatic calls. Conversational models use tools and return session objects; typed decision models return logical, character or numeric vectors with associated model probabilities. Supports the 'Anthropic' <https://platform.claude.com/docs/en/api/overview>, 'OpenAI' <https://developers.openai.com/api/reference/overview> and 'Google Gemini' <https://ai.google.dev/gemini-api/docs> APIs, 'OpenAI'-compatible endpoints, local 'Ollama' <https://ollama.com> models, and the 'Claude Code' and 'Codex' command line tools. Decision models include 'Jev', hosted by 'TypeSafe AI' <https://docs.typesafe.ai/introduction>, and the local 'Clef'. Top-level agent calls and their executed code can be recorded in 'R' scripts, 'R Markdown' and 'Quarto' documents, or 'Jupyter' notebooks. Recorded 'R' scripts can be replayed without model calls when the required inputs and packages are available.

License Apache License (== 2.0)

URL https://broccolito.github.io/gptr/, https://github.com/Broccolito/gptr

BugReports https://github.com/Broccolito/gptr/issues

Depends R (>= 4.2.0)

Imports cli (>= 3.6.2), curl (>= 6.0.0), graphics, grDevices, jsonlite (>= 1.8.8), methods, processx (>= 3.8.6), ps (>= 1.8.0), rlang (>= 1.1.0), stats, tools, utils, yaml (>= 2.3.0)

Suggests bslib, chromote, codetools, data.table, DBI, duckdb, httpuv, keyring, knitr, later, magick, openssl, ragg, reticulate, rmarkdown, RSQLite, rstudioapi, shiny, stringi, testthat (>= 3.2.0), vctrs, withr

Enhances repr

VignetteBuilder knitr, rmarkdown

Config/testthat/edition 3

Copyright file inst/COPYRIGHTS

Encoding UTF-8

Language en-US

NeedsCompilation no

SystemRequirements Optional: Ollama (>= 0.35.1) for native local decision models; other optional external programs: 'claude' (Claude Code) and 'codex' (Codex) for subscription providers, 'quarto' for 'Quarto' documents, and a 'Chrome'-family browser for app screenshots.

Config/roxygen2/version 8.1.1

Author Wanjun Gu [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-7342-7000>), Tianlu Zhu [ctb] (UI/UX polishing and CI/CD), Gianmarco Bellucci [ctb] (Testing, ORCID: <https://orcid.org/0000-0002-2635-1699>)

Maintainer Wanjun Gu <wanjun.gu@ucsf.edu>

Repository https://cran.r-universe.dev

Date/Publication 2026-11-01 07:39:43 UTC

RemoteUrl https://github.com/cran/gptr

RemoteRef HEAD

RemoteSha 9e80ed4c4f9c308532a486b02039381cd86c343a

Contents

as.data.frame.gptr_session_summary 4
format.gptr_risk 4
format.gptr_session 5
gptr-background 6
gptr_adapter 7
gptr_agent 8
gptr_api 9
gptr_artifacts 10
gptr_backend 11
gptr_blocks 12
gptr_cache 12
gptr_cancel 13
gptr_check 14
gptr_coach 15
gptr_command 17
gptr_config 17
gptr_context_block 19

gptr_describe . . . 20
gptr_doc . . . 21
gptr_egress . . . 23
gptr_env . . . 25
gptr_export . . . 26
gptr_fake_provider . . . 28
gptr_fork . . . 29
gptr_hook . . . 30
gptr_init . . . 30
gptr_jobs . . . 31
gptr_login . . . 32
gptr_mcp . . . 33
gptr_mcp_add . . . 34
gptr_mcp_serve . . . 35
gptr_models . . . 36
gptr_on . . . 37
gptr_options . . . 38
gptr_parallel . . . 42
gptr_permissions . . . 44
gptr_plugins . . . 45
gptr_policy . . . 46
gptr_preimage . . . 47
gptr_prob . . . 48
gptr_prompt . . . 49
gptr_prompt_section . . . 50
gptr_provider . . . 51
gptr_providers . . . 52
gptr_redact . . . 53
gptr_register . . . 55
gptr_registry . . . 55
gptr_reload . . . 56
gptr_return . . . 57
gptr_rewind . . . 57
gptr_risk . . . 60
gptr_router . . . 61
gptr_scrub . . . 62
gptr_security . . . 63
gptr_sessions . . . 68
gptr_skills . . . 69
gptr_source . . . 70
gptr_spec . . . 72
gptr_steer . . . 72
gptr_step . . . 73
gptr_tailor . . . 74
gptr_tool . . . 76
gptr_tool_result . . . 77
gptr_trust . . . 78
gptr_usage . . . 79

gptr_wait 80
peter 81
print.gptr_session 86
print.gptr_session_summary 87
str.gptr_session 87
summary.gptr_session 88

Index **89**

`as.data.frame.gptr_session_summary` *Convert a session summary to a data frame*

Description

Convert a session summary to a data frame

Usage

```
## S3 method for class 'gptr_session_summary'
as.data.frame(x, ...)
```

Arguments

`x`	A `gptr_session_summary`.
`...`	Further arguments, such as `row.names` and `optional`, passed to the ordinary data frame method.

Value

An ordinary `data.frame` containing the complete history.

`format.gptr_risk` *Format a risk classification*

Description

Format a risk classification

Usage

```
## S3 method for class 'gptr_risk'
format(x, ...)

## S3 method for class 'gptr_risk'
print(x, ...)
```

Arguments

x	A gptr_risk object.
...	Unused.

Value

format() returns a character vector: the level line, then one line per flagged call (control, bidi and zero-width characters escaped); print() returns x invisibly.

Examples

```
r = gptr_risk("x = 1; unlink('data', recursive = TRUE)")
format(r)
print(r)
```

---

format.gptr_session *Format a session as its text*

Description

Format a session as its text

Usage

```
## S3 method for class 'gptr_session'
format(x, ...)

## S3 method for class 'gptr_session'
as.character(x, ...)
```

Arguments

x	A gptr_session.
...	Unused.

Value

x$text.

gptr-background *Background sessions (experimental)*

Description

`peter(..., background = TRUE)` returns the session at once while its run keeps going. A timer from the **later** package advances the run every 50 ms while the R console is idle, so you can keep working. Piping into the running session, `session |> peter("...")`, queues a steer that the agent receives after its current tool result; `gptr_wait()` waits for the session, `gptr_cancel()` aborts its run and `gptr_jobs()` lists it. Background sessions are **experimental**: they need the **later** package, and they are never used in examples or in CRAN tests. Background work is not an independent R execution thread: an inline R tool occupies the process while it runs. Returning a session does not establish idle progress in an unverified front end; inspect its status and use a blocking wait when needed.

How a background run behaves

- Streams progress at every idle tick. While a blocking gptr call runs (`peter()`, `gptr_wait()`, `gptr_step()` or a console turn), that call advances the background runs as well and the background timer does nothing.
- R tools of a background run execute at an idle tick when `options(gptr.background_tools = "idle")` (the default). The console is busy while such a tool runs; Ctrl-C then opens the pause menu. With `"wait"` they run only inside a blocking gptr call such as `gptr_wait()`.
- A background run never asks a question at the idle console. When it needs an approval or an answer at an idle tick, the action is not performed: the agent is told that the user will be asked, its run stops, and the session moves to status `"waiting"` with a notice. `gptr_wait()` on the session continues it with one more model request, which tells the agent that you are at the console; the agent repeats the call, and you are asked as usual. Other calls and console turns leave it waiting. A message piped into a waiting session is delivered when it continues.
- `gptr_cancel()` stops a running background session. A waiting session has no run to cancel: `gptr_jobs(kill = TRUE)` stops it (together with every other job).
- When an R tool changed objects in your workspace at an idle tick, one notice names them.
- Under Rscript, knitr and Quarto there is no idle console: background runs progress only inside blocking gptr calls.

Support matrix

Front end	Status
Terminal R on macOS	verified: idle ticks, pause menu, pipe steering
Terminal R on Linux	unverified (the same **later** event loop as macOS)
Rscript, knitr, Quarto	no idle ticks by design; progress inside blocking gptr calls
RStudio	unverified (**later** is expected to run callbacks at its idle console)
Positron, Jupyter (IRkernel), Rgui	unverified
Windows consoles	unverified; a timer polls the reactor, so child pipes are served

Options

`gptr.background_tools` `"idle"` (default) or `"wait"`: whether R tools of background runs execute at idle ticks or wait for the next blocking gptr call.

See Also

`peter`, `gptr_wait`, `gptr_cancel`, `gptr_steer`, `gptr_jobs`

---

gptr_adapter *Define a wire adapter*

Description

An adapter turns gptr's request context into one provider's wire format and its stream back into gptr events. HTTP and process adapters give `build()` and `parse()`; `inprocess` adapters give `stream()`, a generator factory; classifier adapters give `classify`.

Usage

```
gptr_adapter(
  api,
  transport = c("http_sse", "http_ndjson", "http_json", "process_jsonl", "inprocess"),
  build = NULL,
  parse = NULL,
  stream = NULL,
  classify = NULL,
  capabilities = list()
)
```

Arguments

`api`	The wire api this adapter implements (the name providers bind to).
`transport`	`"http_sse"`, `"http_ndjson"`, `"http_json"`, `"process_jsonl"` or `"inprocess"`.
`build`	`function(model, context, opts)` returning a request spec.
`parse`	`function(model, opts)` returning a stream normalizer.
`stream`	`function(model, context, opts)` returning a generator (`inprocess`).
`classify`	Named list for classifier adapters: `build(model, state, questions, opts)` and `parse(model, status, headers, body, questions)`, or `run(model, state, questions, opts)` for `inprocess`. Classifier response parsing has a separate signature from the ordinary chat stream `parse(model, opts)`.
`capabilities`	Named list of adapter capabilities.

Value

A spec of class `c("gptr_adapter", "gptr_spec")`.

Examples

```
gptr_adapter("echo", transport = "inprocess",
             stream = function(model, context, opts) {
               state = new.env()
               state$done = FALSE
               function() {
                 if (state$done) return(NULL)
                 state$done = TRUE
                 list(events = list())
               }
             })
```

`gptr_agent` *Define a sub-agent*

Description

An agent definition names a specialist for `peter(agents = ...)`: its model, tools, skills, system text, backend, preset and limits. `model` and `skills` may be bare identifiers; they are stored unevaluated (as written) and resolved by the gateway. `gptr_agent("name")` alone (or with only `file`) loads a saved definition. Package code should pass strings.

Usage

```
gptr_agent(
  name = NULL,
  description = NULL,
  model = NULL,
  tools = NULL,
  skills = NULL,
  system = NULL,
  backend = c("auto", "inline", "worker", "cli"),
  preset = "minimal",
  max_turns = NULL,
  mode = NULL,
  objects = NULL,
  export = NULL,
  returns = NULL,
  file = NULL
)
```

Arguments

`name`	Agent name.
`description`	One line describing the specialist.
`model`	Model reference: a string, a bare identifier or a provider spec.
`tools`	Tool names (or specs) the agent may use.

skills	Skills to preload.
system	System text of the agent.
backend	"auto", "inline", "worker", "cli" or a registered backend name.
preset	Tool preset of the agent's session.
max_turns	Turn limit, or NULL.
mode	Permission mode ("plan", "manual", "edits", "auto", "yolo"), or NULL to inherit; a sub-agent never runs looser than its caller.
objects	Names of objects the agent may see.
export	Names of objects the agent returns. Keep them self-contained: a worker agent sends back only these objects, so an exported function or plot finds the caller's objects but none of the agent's others.
returns	JSON Schema of a structured answer, or NULL.
file	Path of an agent definition file.

Value

A spec of class c("gptr_agent", "gptr_spec").

Examples

```
gptr_agent("stats", description = "Statistical reviewer", model = "anthropic/claude-opus-5-5",
           skills = "statistics")
```

---

gptr_api *Extension API version and features*

Description

The version of gptr's public extension API (independent of the package version) and the features this build offers: kind.<name> for every registered kind, event.<name> for every cataloged event, and the named features lazy_activation, declarations, ctx.decide, ctx.secret, route and services. Plugins test features instead of comparing versions.

Usage

```
gptr_api()
```

Value

A gptr_api list with version (a package_version) and features (character).

Examples

```
gptr_api()$version
"kind.router" %in% gptr_api()$features
```

gptr_artifacts *List, open, relaunch and stop artifacts*

Description

Artifacts are Shiny apps the agent builds from objects in your session. It writes `.gptr/artifacts/<id>/app.R` and calls `peter$app("<id>", data = c("obj"))`, which snapshots the named objects into an immutable version directory (`v001/`, `v002/`, ...) and, when a person is at the console or with `launch = TRUE`, serves it from a supervised background R process on `127.0.0.1`, so the console stays free. `gptr_artifacts()` lists the artifacts of the workspace (`.gptr/`, else a temporary directory) and manages their processes.

Usage

```
gptr_artifacts(id = NULL, open = FALSE, stop = FALSE, version = NULL)
```

Arguments

`id`	`NULL` to list every artifact, or the id of one artifact.
`open`	`TRUE` to open the artifact in the IDE viewer or the browser (only in an interactive session); a stopped artifact is relaunched first.
`stop`	`TRUE` to stop the artifact's process (interrupt, a 3 s grace, then the process tree is killed).
`version`	The number of a stored version (`2` for `v002/`) to relaunch and make current.

Details

The built-in `kind = "html"` uses a working `page.html` and a Shiny serving wrapper. Both built-in kinds need Shiny to launch. `peter$app(..., launch = FALSE)`, or the default in a script, knitr or tests (its messages say how to start it), creates a version without launching it; startup and browser checks therefore remain unperformed. Browser session checks and screenshots additionally need chromote and Chrome. A check value of `NA` means it was skipped. Separate processes organize execution but are not OS sandboxes.

Value

Without `id`, a `gptr_artifacts` data frame with the columns `id`, `title`, `version` (the current version), `status` (`running`, `stopped` or `failed`), `url`, `pid`, `bytes` (disk use of the artifact directory) and `path` (the working copy). With `id`, the `gptr_artifact` handle: a list with `id`, `title`, `kind`, `version`, `url` (it carries the launch's access token), `path`, `status`, `checks` (`parse`, `launch`, `http` and `session`, each `TRUE`, `FALSE` or `NA`, plus `messages`), `screenshot` and `session`; it is returned invisibly after `open`, `stop` or `version`.

Security considerations

An artifact runs model-written code in a separate R process whose environment holds none of your registered secrets. It listens only on `127.0.0.1`, and every launch gets a new 128-bit access token that its URL carries (`?gptr_token=`); without the openssl package on Windows there is no token and a notice says so. The data snapshot in `vNNN/data/` is a copy of the objects named in `data`; the `.gptr/.gitignore` that `gptr_init()` writes keeps it and `run/` out of git.

Options

`gptr.artifact_max_bytes` (default `5e8`): the largest total size of the objects one version may snapshot (each part counted once, as in the workspace); above it `peter$app()` signals `gptr_error_artifact_too_large`.

See Also

`vignette("tools-and-artifacts", package = "gptr")` for working files, data snapshots, checks and lifecycle examples.

Examples

```
gptr_artifacts()                    # an empty listing when no artifact exists
```

---

gptr_backend *Define a sub-agent backend*

Description

A backend runs child sessions: `start(spec, ctx)` returns a handle the reactor can poll, `cancel(handle)` stops it and every process it started. Backends never block the reactor for more than 50 ms.

Usage

```
gptr_backend(name, start, poll = NULL, cancel, capabilities = list())
```

Arguments

`name`	Backend name, usable as `backend = <name>`.
`start`	`function(spec, ctx)`.
`poll`	`function(handle)` or `NULL`.
`cancel`	`function(handle)`.
`capabilities`	Named list: `parallel`, `live_objects`, `ask`.

Value

A spec of class `c("gptr_backend", "gptr_spec")`.

Examples

```
gptr_backend("echo", start = function(spec, ctx) NULL, cancel = function(handle) NULL)
```

gptr_blocks *List the agent blocks of a document*

Description

Reads an `.R`, `.Rmd`, `.qmd` or `.ipynb` document and lists its gptr blocks with their status: `fresh` (the owning call's prompt matches), `stale` (the prompt changed, or no call owns the block), `user-edited` (the body no longer matches its `sha`) or `undone` (made inert by a rewind). Interpolated values are only known when the call runs, so they are not checked here. It reads the document; the only write is that blocks an `Rscript` run queued before it was killed are applied first (never over your edits).

Usage

```
gptr_blocks(file)
```

Arguments

`file`	Path of the document.

Value

A `gptr_blocks` data frame with columns `id`, `lines`, `prompt`, `status`, `model`, `date`, `tokens`, `cost`, `session`. `cost` is in USD: `0` for a known free route (a local model), `NA` when unknown or not recorded.

Examples

```
f = tempfile(fileext = ".R")
writeLines(c('peter("add one")',
             "# >>> gptr:7f3a21 model=fake/fake-1 date=2026-09-29 prompt=3b1c9a0e77d2",
             "x = 1 + 1", "# <<< gptr:7f3a21"), f)
gptr_blocks(f)
```

gptr_cache *Inspect, prune or clear gptr's caches*

Description

`info` lists the System 1 answer cache (held in memory for the R session while there is no `.gptr/`), the System 2 answer cache used for replay, temporary spill files and unapplied document sidecars (deferred or pending blocks; a second row lists those of documents outside any `.gptr/`, kept in the user cache; they are never removed here: apply them with `gptr_doc(path, sync = TRUE)`). `prune` removes System 2 answers whose block is neither in its document nor queued for it in a sidecar, System 1 answers unused for 90 days, temporary files older than `getOption("gptr.spill_days", 7)` days and older refreshed model catalogs; `clear` removes a kind entirely.

Usage

```
gptr_cache(
  action = c("info", "prune", "clear"),
  kind = c("all", "s1", "s2", "tmp")
)
```

Arguments

action	"info", "prune" or "clear".
kind	"all", "s1", "s2" or "tmp".

Details

Clearing the System 2 answer cache does not remove the R code of document blocks, but saved answers needed to reconstruct a replayed call may no longer be available.

Value

info: a gptr_cache_info data frame with columns kind, entries, bytes, oldest, path. prune and clear: the number of entries removed, invisibly.

Examples

```
proj = tempfile("proj")
dir.create(file.path(proj, ".gptr"), recursive = TRUE)
old = options(gptr.project_root = proj)
gptr_cache()
gptr_cache("prune", "tmp")
options(old)
unlink(proj, recursive = TRUE)
```

---

gptr_cancel *Cancel running sessions*

Description

Aborts the run of a session (or of each session in a list): transfers are canceled, child processes stopped, the partial turn recorded as aborted, queued items moved to dropped, and the status set to aborted. Idle sessions are left alone.

Usage

```
gptr_cancel(x)
```

Arguments

x	A session or a list of sessions.

Value

x, invisibly.

Examples

```
session = peter("long task", model = gptr_fake_provider(list(list(hang = TRUE))),
  .run = FALSE)
gptr_cancel(session)
```

---

gptr_check *Check an extension for conformance*

Description

Runs gptr's conformance suite, offline, in a scratch registry. For a spec: its fields and kind validator, schema validity, direct tool descriptions of at most 400 tokens, prompt-section budgets, empty-input handling of tools, the permission matrix for policies, start and cancel for backends, and wire-fixture replay for adapters when the adapter checker is available. For a factory `function(gptr)`: it loads, registers something, takes no action at load, and each registered spec passes. For an installed package name: its `inst/gptr/plugin.json` manifest, API requirement, factory, every provided capability, and bare gptr identifiers in its code.

Usage

```
gptr_check(x, error = FALSE, tokens = FALSE)
```

Arguments

x	A spec, a factory `function(gptr)`, or the name of an installed plugin package.
error	TRUE to signal gptr_error_conformance when a check fails.
tokens	TRUE to also report declaration costs and the printed cost of each member's examples.

Value

A gptr_check data frame with columns target, check, ok and message.

Examples

```
gptr_check(gptr_tool("add", "Add two numbers",
                     parameters = list(type = "object", required = I(c("a", "b")),
                                       properties = list(a = list(type = "number"),
                                                         b = list(type = "number"))),
                     fun = function(a, b) a + b, exposure = "r", namespace = "demo"))
```

gptr_coach *Coach a harness by running your task*

Description

gptr_coach() improves the *coaching* of a harness (the note, skills, review and rules that gptr gives peter; see gptr_tailor()) by running your task. Your starting harness does the task once; two coaches read that run, one about how peter works and one about what an expert in the subject knows, and each proposes new coaching; each proposal does the task once; a third coach, the composer, compares the runs and composes the coaching to keep; and that coaching does the task once more. Code then decides whether to keep it: the final run must end normally, deliver at least what the starting run delivered, take no riskier kind of action, meet every require rule and pass your check. Every session runs unattended in its own R process, with copies of your objects and an empty folder of its own, so a step that needs your approval ends that run; most tasks need mode = "auto".

Usage

```
gptr_coach(
  ...,
  harness = peter,
  check = NULL,
  coach = NULL,
  k = 1L,
  strategy = "parallel",
  focus = c("procedure", "domain"),
  surface = c("note", "skills", "review", "rules")
)
```

Arguments

...	What you would give the harness: the prompt, your objects, and any peter() arguments such as mode, model, budget or tools. They apply to every run of this call and are not kept in the result; budget, else the starting harness's, also caps each coach session and the judge. Each run is one unattended session, so parallel, choices, levels, threshold, min_confidence, uncertain, background, .run, .stdin and envir are refused.
harness	The harness to start from: peter or a harness.
check	NULL or your own test of a good result: a function of r, a list with text, status, value and objects of the final run, that returns TRUE or FALSE. It runs in this R session only, and no coach sees it.
coach	The model of the coaches, a name or a provider spec; NULL uses the runs' model.
k	How much coaching, 1 to 5. With "parallel": k starting runs, k coaches of each focus, one probe of each, and k composers that each see every probe; when more than one of their final runs passes, a blind judge, which sees only the task and what each final delivered, picks one. With "sequential": k starting runs,

	then k rounds, each building on the composition of the round before and seeing every earlier round.
strategy	"parallel" or "sequential" (see k). It is not called mode, so that mode = "auto" still reaches peter.
focus	Every focus to run, one proposing coach each: "procedure" (how peter works), "domain" (what an expert knows), or your own words.
surface	The parts of the coaching that may change, among "note", "skills", "review" and "rules". Only the procedure coach writes rules.

Details

Before it starts, gptr_coach() asks the usual question about sending your data to each model, while you are there, and says how many sessions it will run: k * (2 + focuses) runs of your task and k * (1 + focuses) coach sessions, plus up to one judge session when k > 1; with the defaults, 4 runs and 3 coach sessions. On a Claude Code or Codex plan they use your plan's limits, at most two at once; when a plan limit is reached, it starts nothing new and returns your starting harness. A call recorded in a document (h = gptr_coach(...) at the top level of a script, R Markdown, Quarto or Jupyter document) replays from its block with no model, no key and no process.

Value

A harness: your starting harness with the new coaching when it was kept, otherwise your starting harness. Either way it has a new id and a record (h$record: each run's outcome, turns, tokens, cost, highest risk and unmet requirements, and the decision with its reason; no data from your runs). h$report holds the full report, with every answer, value, transcript and coach reply, in this R session only. A call replayed from its document block returns the recorded coaching with an empty record; the block's ## Decision: line holds the decision.

See Also

gptr_tailor() to build or change a harness by hand, and peter().

Examples

```
# A run that is wrong unless its note says how to treat missing values, and a coach that
# writes that note (both scripted; this starts 7 R processes and takes several seconds)
solver = gptr_fake_provider(function(request) {
  if (length(request$last_results)) return("Done.")
  fix = grepl("na.rm", paste(unlist(request$system), collapse = " "), fixed = TRUE)
  list(tool = "r", input = list(code = if (fix) "gptr_return(mean(x, na.rm = TRUE))" else
    "gptr_return(mean(x))"))
}, name = "solver")
coach = gptr_fake_provider(list(paste0(
  '{"note": "Drop missing values with na.rm = TRUE before you average.", ',
  '"notes": "The starting run returned NA."}')), name = "coach")

x = c(1, NA, 3)
mean_peter = gptr_coach("Average x.", x, model = solver, mode = "auto", coach = coach,
                        check = function(r) isTRUE(r$value == 2))
mean_peter$record$decision
```

gptr_command *Define a slash command*

Description

A console command `/name args`: `handler(args, ctx)` gets the raw text after the name and returns `NULL`, a character vector (printed) or `list(prompt = "...")` (sent as a prompt).

Usage

```
gptr_command(name, handler, description = NULL, complete = NULL)
```

Arguments

`name`	Command name without the leading `/`.
`handler`	`function(args, ctx)`.
`description`	One line for `/help`.
`complete`	Optional `function(prefix, ctx)` returning completion candidates. Stored as metadata for frontend integrations; the built-in console does not currently invoke it.

Value

A spec of class `c("gptr_command", "gptr_spec")`.

Examples

```
gptr_command("rows", function(args, ctx) paste("rows:", nrow(mtcars)), description = "Show rows")
```

gptr_config *Show or change gptr's settings*

Description

With no arguments, returns the effective settings with the layer each value came from: package defaults < user `settings.json` < project `.gptr/settings.json` (an untrusted project only tightens `mode`, `context`, `record`, permission rules and local-only inference; a trusted one also sets `mode`, while `context` and `record` still only tighten) < the user-level project file < `options(gptr.*)` < this R session. With named arguments, sets those keys in one scope.

Usage

```
gptr_config(..., .scope = NULL)
```

Arguments

...	Named settings, e.g. mode = manual, model = sonnet, budget = list(cost = 2).
.scope	NULL (the project when a .gptr/ workspace exists, else this session), or "session", "project" or "user".

Details

model, small_model, system1, mode and preset take bare names (mode = plan); NULL removes a key from the scope. egress (the providers you allow automatic context to go to) is accepted only at user scope. mode = yolo and turning off local-only Ollama inference (providers = list(ollama = list(local_only = FALSE))) are accepted only at user or session scope; a project can only keep local-only inference on. Called from model code during a run, it is refused.

The "session" scope means this R process, not one conversation. User settings live at tools::R_user_dir("gptr", "config")/settings.json; project settings live in .gptr/. A no-argument call always reads effective values, even when .scope is supplied. Inspect attr(gptr_config(), "sources") to see their origins. Named lists merge with lower layers; protected objects such as providers must be set as a whole named list. A plugins entry starting with ./ or ../, like a relative skills.paths entry, is found from the project root, not from R's working directory.

Value

Without arguments, a gptr_config list. Otherwise the previous values in that scope, invisibly (a named list; NULL where the key was unset).

Choosing controls

Use settings for defaults such as model, mode, context, budget, record, replay and registered extension selections. Use gptr_options for transport timeouts, output caps, undo retention and concurrency. Arguments to peter() choose values for one call, within inherited safety restrictions. vignette("configuration", package = "gptr") groups all core settings and options and explains trust and scope.

See Also

gptr_init(), gptr_trust(), gptr_permissions() and gptr_options.

Examples

```
old = gptr_config(mode = plan, .scope = "session")
gptr_config()$mode
gptr_config(mode = old$mode, .scope = "session")
```

`gptr_context_block` *Define a context block*

Description

A block of context the model receives in user messages: `provide(ctx, budget)` returns text (or `NULL` to skip). `placement` puts it in the first message, in later turns, or both; turn blocks equal to their previous text are skipped. `authority = "operator"` is accepted only from user, plugin and built-in records.

Usage

```
gptr_context_block(
  name,
  provide,
  placement = c("turn", "first", "both"),
  authority = c("data", "operator"),
  budget = 300L,
  order = 650L
)
```

Arguments

`name`	Block name (the tag it is wrapped in).
`provide`	`function(ctx, budget)`.
`placement`	`"turn"`, `"first"` or `"both"`.
`authority`	`"data"` or `"operator"`.
`budget`	Token budget; longer text is truncated.
`order`	Position among blocks.

Value

A spec of class `c("gptr_context_block", "gptr_spec")`.

Examples

```
gptr_context_block("lab_notebook", function(ctx, budget) "Experiment 12: cohort B only.",
                   placement = "first", order = 650L)
```

gptr_describe *Compact, budgeted description of an R object*

Description

An S3 generic used for the <attached> and <workspace> context blocks, peter$describe() and console mentions. Methods return successively richer levels of detail and the richest level whose estimated size fits budget is returned. Methods never force promises, never do I/O (no dbListTables(), no collect()) and never call str() on the object. Packages add methods with a delayed S3method(gptr::gptr_describe, <class>).

Usage

```
gptr_describe(x, budget = 150L, ...)

## Default S3 method:
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'data.frame'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'matrix'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'list'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'environment'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'gptr_session'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class '`function`'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'formula'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'lm'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'dgCMatrix'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'Seurat'
gptr_describe(x, budget = 150L, ..., level = NULL)
```

```
## S3 method for class 'SingleCellExperiment'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'ArrowTabular'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'Dataset'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'DBIConnection'
gptr_describe(x, budget = 150L, ..., level = NULL)

## S3 method for class 'ggplot'
gptr_describe(x, budget = 150L, ..., level = NULL)
```

Arguments

x	Any R object.
budget	Estimated tokens, at least 20 (default 150).
...	Passed to methods; built-in methods accept `level` (1 = header only; larger values give more detail).
level	Built-in methods: the detail level to return (default: the richest that fits).

Value

A character vector of lines; the first is a header `<class> shape, size`.

Examples

```
gptr_describe(mtcars, budget = 60)
gptr_describe(as.Date("2026-01-01") + 0:9)
```

---

gptr_doc *Bind this R process to a history document*

Description

`gptr_doc(path)` binds a history document to this R process and is your explicit consent that gptr writes agent blocks into it. Calls written in that document (run with `source()`, `gptr_source()`, `Rscript`, knitr or Jupyter) get their blocks below them. Calls in no document (the console, `Rscript -e`, standard input) are appended to it, except to a notebook, as `s_<id> = peter("prompt", <context names>)` (later turns of that session as `s_<id> |> peter(...)`). The statement ends with `model = "<provider/model>"` or `mode = "<mode>"` when its turn ran with a model or mode the statement would not get without them, as after `/model`, `/mode`, the plan menu or in a console started as `peter(mode = plan)`: for `=`, one other than the `model` and `mode` settings (the model is left out when the call started the session without naming one); for `|>`, one other than the session's previous turn in the transcript. A model

is written as its full reference (`model = opus` is recorded as `"anthropic/..."`). A model given as a provider object is not recorded on a session's first statement. Your variable name and the other arguments are not recorded. A call in another document is recorded only into that document, with its own consent; a run that cannot ask for it records nothing and says so once per document. Nothing is written until a block is recorded. `gptr_doc(FALSE)` removes the binding and `gptr_doc()` shows it. `sync = TRUE` writes the blocks that were recorded while the document could not be written, and says how many: a Jupyter notebook that was open (run it from another R session after closing the notebook), or the deferred writes of an `Rscript` run that was killed. Such blocks wait in the `.gptr/` of the document's project, else in `tools::R_user_dir("gptr", "cache")`, so any R process of the same user can apply them.

Usage

```
gptr_doc(path = NULL, format = NULL, sync = FALSE)
```

Arguments

`path`	`NULL` to return the current binding, `FALSE` to remove it, or the path of an `.R`, `.Rmd`, `.qmd` or `.ipynb` document.
`format`	`NULL` (from the file extension) or the format of that extension: `"r"`, `"rmd"`, `"qmd"` or `"ipynb"`; `"transcript"` records console turns into an `.R` file as one steered session.
`sync`	`TRUE` applies the pending blocks of the document now and reports how many it wrote.

Details

Eligible top-level calls and pipe chains own document blocks; calls nested inside loops or functions do not. Only blank lines may separate a call's statement from its blocks: after a comment or code line between them the blocks belong to no call, so the call is asked again and gets a new block while the old block's code still runs. A block records the R code the agent ran: of a call that stopped part-way, the expressions that ran, and code that changed your objects even where the model passed `record = false`, which leaves out only inspection. Binding supplies recording consent even when automatic recording is off, but `.opts = list(record = FALSE)` suppresses recording for that call. Binding does not reconstruct earlier unrecorded turns. A narrative summary or model-generated script draft needs review and execution checks before it can serve as a reproducible workflow.

In a console transcript (an `.R` file) a turn's block opens with `#<` comment lines holding what the console sent with the prompt: the head of each `@file` (at most 40 lines, redacted) and the notes of earlier `!` lines. Only that run, under the turn's `s_<hex>` statement, is sent again, and none from a transcript whose gptr markers are malformed; a line of recorded code that would continue the run gets one more # (`##<`). In every format a whole comment line of recorded code that reads as a `## Steer:` or `## Follow-up:` line or as a block marker gets one more # too, so code the model wrote is never sent as your context or your steer and never ends its block. The block's header names the objects the turn created (`created=`), as the header of each direct R line does (`# direct R (no model) created=p`), and a comment line of `!` code that reads as a console line, such as `# /tmp is scratch`, or as a block marker gets one more #. A block whose code holds a marker line inside a string is not written, and gptr says so. After each `/undo`, `/redo`, `/retry` and `!` line the transcript is read again from the top: a direct R line that uses an object

last created above it by an undone turn or by a line that is off is turned off (`#~` before each line, `# direct R (no model; uses undone fit)`) and back on when that turn returns, so `source()` does not stop there; a whole `fit = <value>` makes `fit` and uses only what `<value>` names. The console names the lines it turned off. Your own objects stay in your session: an undo removes what the turn created and restores what it changed. So a later turn that uses an object of a line that is off still stops `source()`, as turn blocks are never turned off, and a direct R line that read an object the turn changed, such as `n = nrow(dat)` after the turn cut `dat`, stays on and replays with the restored value. Names are found with `all.names()`: a column or formula term with the same name can turn a line off, and a name reached through a string, as in `get("fit")`, is not seen.

Value

With `path = NULL`, the binding `list(path, format)` or `NULL`, visibly. Otherwise the previous binding, invisibly.

Options

Options of history documents (`?gptr_options` collects every option): `gptr.record` (settings, `"ask"`): `"auto"` records into documents without asking, `"off"` prevents automatic recording; an explicit binding supplies consent. Also the settings key `record`, which a project can only tighten. A call's `.opts = list(record = FALSE)` opts out even when bound; `.opts = list(record = TRUE)` retains the usual consent checks. `gptr.replay` (settings, `"auto"`): the replay mode `"auto"`, `"replay"`, `"live"` or `"record"` (also the `GPTR_REPLAY` environment variable); a call's `replay =` argument wins. `gptr.doc_output_lines` (12): `#>` output lines recorded per execution. `gptr.doc_source_frames` (`TRUE`): `FALSE` stops gptr from reading `source()` frames to find calls in scripts sourced without srcrefs. `gptr.spill_days` (7): `gptr_cache("prune")` removes temporary files older than this.

Examples

```
f = tempfile(fileext = ".R")
writeLines("library(gptr)", f)
gptr_doc(f)
gptr_doc()
gptr_doc(FALSE)
```

`gptr_egress` *Data egress: what gptr sends to model providers*

Description

gptr sends data to a model provider only when you run a prompt with that provider's model, or when model code that the permission mode allowed starts a sub-agent with that model. The package itself collects nothing: there is no telemetry, no usage reporting and no network access when the package loads. This page lists what a request contains so that you can decide what may leave your machine.

Details

R tools compute on objects already in memory; a large dataset does not need to be uploaded in full for a conversational agent to analyze it. Descriptions and tool results can still contain data values. Native typed decision inputs use the separate encoding described below.

What a request contains

- Your prompt and the conversation so far.
- With summary context, a budgeted description of each attached object (its class, dimensions, column names and types and a few values); a small object can appear in full.
- Automatic context: a short listing of the objects in the evaluation environment (names, classes and sizes); the objects you added, changed or removed and the top-level R expressions you ran since the previous prompt (with the output of `!code` lines in the `peter()` console); the R version, platform and available packages; and the project instruction files (`AGENTS.md`, `CLAUDE.md`, `.gptr/vignette.Rmd`).
- Tool results: the printed output of R code the model ran, the parts of files it read, and images of plots it made. In `manual` mode every action that is not known to be read-only needs your approval before its result can be sent.
- With `gptr_coach()`, each coach gets the task, the current coaching and, for each run, its answer, a summary of its value and objects, the text of the files it wrote and its transcript, up to 50,000 characters each; the judge gets the task and what each final run delivered. The coach model can differ from the runs' model.

A System 1 question (a classifier model such as Jev) sends the inputs themselves: each element of a vector (text up to 100,000 characters), each row of a data frame as a record, a description of larger objects, and for a piped session its last answer (at most `gptr.s1_state_max` characters). Do not ask System 1 questions about data that may not leave your machine.

Registered secrets, and their URL-encoded, JSON-escaped and base64 forms, are replaced by markers before anything is sent; a copy the code changes in another way, such as reversed or cut, is not recognized.

Limiting automatic context

`.opts = list(context = "names")` sends workspace object names, the workspace changes and attached-object headers, and `.opts = list(context = "none")` suppresses the workspace listing, the workspace changes and attached-object descriptions. Prompts, tool results and native typed decision inputs have their own paths. Attach only the objects the task needs. The `plan` mode permits actions classified as read-only so you can inspect a proposed workflow before allowing changes to your workspace.

Acknowledgment

The first time you use a provider (or a command-line route such as `claude_code` or `codex`) in an interactive session, gptr shows what is sent and records your acknowledgment in your user settings. A non-interactive run with a provider that was never acknowledged stops with a `gptr_error_egress` condition. Record the acknowledgment once with `gptr_config(egress = list(anthropic = "ack"), .scope = "user")` (this replaces the stored list, so name every provider you use). Acknowledgment

is required even with `.opts = list(context = "none")`: prompts, tool results, R-session information and project instructions can still leave the machine. Offline providers such as `gptr_fake_provider()` and local providers on a loopback address need no acknowledgment. An Ollama server can forward requests to cloud models, so Ollama is exempt only while its local-only inference is on (the default; see `providers` in `gptr_config()`).

Other network use

gptr contacts nothing else on its own. The model catalog is refreshed only by `gptr_models(refresh = TRUE)`, reachability checks run only with `gptr_providers(check = TRUE)`, and MCP servers, sub-agents and command-line tools are contacted only when a session uses them. The `claude` and `codex` command-line routes send the conversation through those vendors' tools under their own terms.

See Also

gptr_security, gptr_options, `gptr_providers()`, `gptr_config()`.

---

gptr_env *Load keys from a .env file*

Description

Reads a `.env` file with gptr's own parser (a UTF-8 byte-order mark, CRLF line ends, `export` prefixes, single and double quotes, multi-line quoted values, `#` comments and hyphenated names such as `jev-key` are understood), maps aliases onto canonical variable names (`jev-key`, `JEV_KEY`, `JEV_API_KEY` and `TYPESAFE_KEY` become `TYPESAFE_API_KEY`; a line spelled exactly as the canonical name wins over an alias), registers secret values in gptr's vault for redaction, and exports the canonical names with `Sys.setenv()`. Plain configuration fields with suffixes such as `_URL` or `_PATH` are not registered as secrets. It never prints, logs or returns a value.

Usage

```
gptr_env(
  path = ".env",
  aliases = NULL,
  set_env = getOption("gptr.env_export", TRUE),
  override = FALSE,
  quiet = FALSE
)
```

Arguments

`path`	Path of an existing `.env` file.
`aliases`	`NULL`, or a named list of extra aliases, `canonical = c("alias", ...)`, added to the built-in table and to the registered `env_alias` specs.

set_env	If TRUE, export the canonical names to the environment of this R process; if FALSE, keep the values in the vault only. Defaults to getOption("gptr.env_export", TRUE).
override	If TRUE, overwrite variables that are already set.
quiet	If TRUE, do not report what was loaded.

Value

Invisibly, a gptr_env_report data frame with columns name (as spelled in the file), variable (the canonical name), secret, fingerprint and action ("set", "registered", "skipped" or "duplicate"), and the attribute bad_lines (the numbers of malformed lines). It holds names and fingerprints, never values.

Examples

```
f = tempfile(fileext = ".env")
writeLines("example-token=example-not-a-real-token-123", f)
gptr_env(f, set_env = FALSE)$variable
unlink(f)
```

---

gptr_export *Export a recorded R document as ordinary R or a clean Peter script*

Description

Reads a synchronized .R history without executing it, asking a model or recovering pending writes. In "r" format the original Peter statements become comments and their recorded R stays executable. In "peter" format calls stay active and recorded R becomes comments, so it is not executed a second time. User R and comments keep their source order.

Usage

```
gptr_export(file, to, format = c("r", "peter"))
```

Arguments

file	An existing .R document with fresh recorded blocks.
to	A new .R path in an existing directory. Existing paths are never replaced.
format	"r" (default) or "peter".

Details

Only bare or symbol-assigned literal Peter calls and Peter-only native pipelines are admitted; a pipeline may start from a session handle an earlier Peter statement assigned, as the later turns of a console transcript do, and after its prompt a call may name objects without argument names, as a console turn's `@objects` are written. Every named argument must be one literal (a string, number, logical or `NULL`), so a call with `agents = list(...)`, `budget = list()` or `envir = new.env()` is refused, and the refusal names that argument. A harness definition, `<name> = gptr_tailor(...)`, is ordinary code. Distinct statements sharing a line (including semicolon-separated statements), mixed pipelines, nested or computed calls, pending writes, missing or edited blocks, secret markers and interpolated-input hashes are refused. An undone block that no statement owns, a console turn that `/undo` or `/retry` undid under its `#~` statement, is left out. Code that calls a `peter$` member, such as `peter$sh()`, exports as ordinary R; code that calls `peter()` is refused. Raw exports also refuse ordinary code that reads a session handle after its Peter statement and before ordinary code assigns that name again, and a function that names any session handle of the document, wherever it is defined, as it reads the name when called. The Peter format retains such ordinary references, since its calls remain active. A top-level `gptr_doc()` whose string literal names the source (relative to the source's directory or the working directory) becomes a comment in raw R and binds the destination in the Peter format, so its calls record there: in the source's directory only the file name changes, elsewhere the literal becomes the destination's full path. It must be alone on its lines. Other `gptr_doc()` calls are kept. Feedback follows each pipeline operand when calls end on separate lines and only its pipe operator or a comment follows the closing call. Feedback follows the next operand's indentation (at most 74 characters); other layouts keep feedback after the complete statement. Peter calls already show their prompts, so only raw R adds prompt comments. Long raw prompt comments wrap at eighty characters. Strip the recorded-prompt prefix from the first line and the two-character comment prefix from continuations, then join without separators to recover the complete escaped R string literal. A Peter-only pipeline on one line keeps grouped feedback after the statement. Call source is never reformatted.

A completed-turn identity and matching source hashes establish a recorded snapshot, not verified execution or a complete session. Ordinary console input and `record = FALSE` effects cannot be reconstructed. External files, objects and package versions must still be supplied. Remaining user code may use gptr or perform network and filesystem operations. Captured `#>` output is limited to twelve lines per block and eighty characters per line; outputs are comments. Neither format keeps a console turn's `#<` lines (what the console sent with its prompt, see `gptr_doc()`), so an exported call is asked without them. Encoding, line endings and the final newline follow the input file. A UTF-8 byte order mark is preserved; source that exported file with `source(to, encoding = "UTF-8-BOM")`.

Value

Invisibly, the normalized destination path.

See Also

`gptr_doc()`, `gptr_source()`, `gptr_blocks()`

Examples

```
input = tempfile(fileext = ".R")
```

```
output = tempfile(fileext = ".R")
writeLines(c("x = 1", "# Keep ordinary code and comments.", "y = x + 1"), input)
gptr_export(input, output)
readLines(output)
unlink(c(input, output))
```

`gptr_fake_provider` *A scripted, offline fake model provider*

Description

`gptr_fake_provider()` returns a provider spec that plays a script instead of calling a model. It needs no network and no keys, so examples, tests and documents can run gptr end to end offline. Use it as `model = <spec>` or register it with `gptr_register()`.

Usage

```
gptr_fake_provider(script, name = "fake", type = c("chat", "classifier"))
```

Arguments

`script`	A list of replies or a function (see Details).
`name`	Provider id: lower-case letters, digits and `-`.
`type`	`"chat"` (the default) or `"classifier"`.

Details

A chat script is a list of replies (request `i` gets reply `min(i, length(script))`, so the last reply repeats) or a function of the request returning one reply. The request is a list with `n`, `model`, `system`, `tools`, `messages`, `last_user`, `last_results` and `params`. A reply is a string (the answer text) or a list with any of: `text`; `thinking` and `signature`; `tool` with `input` and `id` (one tool call); `tools` (a list of `list(name, input)` for parallel calls); `stop` (`"length"`, `"refusal"`, `"pause"` or `"stop"`); `error` with `status` and `after` (fail after `after` text deltas); `overflow = TRUE`; `hang = TRUE`; `json` (a structured answer); and the modifiers `delay`, `gap`, `chunk` and `usage`.

A classifier script (`type = "classifier"`) answers System 1 questions: a list of answers (recycled the same way, one answer per question asked) or `function(state, question)`. An answer is a probability for yes/no questions, a named probability vector for choices, a vector of level probabilities for scores, or `list(error =, status =)`. Classifier results use the provider-neutral decision shape and report calibration as unknown (`NA`).

Value

A provider spec: a list of class `c("gptr_provider", "gptr_spec")` with `id = name`, `api "fake"` or `"fake-classifier"`, one model (`"<name>/<name>-1"`, or `"<name>/<name>-s1"` for classifiers), `offline = TRUE`, the `script`, and `log`, an environment whose `requests` element lists every request the provider received.

Examples

```
fake = gptr_fake_provider(list(
  list(tool = "r", input = list(code = "n = nrow(d)")),
  "There are 32 rows."
))
fake$models[[1]]$ref
fake$offline

judge = gptr_fake_provider(list(0.93), name = "judge", type = "classifier")
judge$models[[1]]$ref
```

---

gptr_fork *Fork a session*

Description

Creates a new idle session whose transcript is the source path up to a cut, with the same entry ids. Nothing live is shared (listeners, queue, run, lock, usage); the fork's file is written lazily at its first own message, with `parentSession` and `gptr.forkOf` in its header. The fork's workspace is an overlay of the source's kept home: it reads every object of the source at zero cost, and its ordinary assignments stay in the overlay, but code that modifies an object by reference (data.table `:=` and `set*()`, environments and R6 objects, `<<-`, `assign(envir =)`) changes the source's object too; copy such objects first, for example with `data.table::copy()`, when a branch must not touch them.

Usage

```
gptr_fork(s, at = NULL, envir = c("overlay", "shared"))
```

Arguments

`s`	A `gptr_session`: the source.
`at`	`NULL` (the last closed boundary; a running source is cut there), an integer `k >= 0` (the end of turn `k`; `0` is an empty conversation) or an entry id.
`envir`	`"overlay"`: the fork evaluates in `new.env(parent = <source home>)`; `"shared"`: the same home.

Value

A new idle `gptr_session`.

Examples

```
fake = gptr_fake_provider(list("A", "B"))
session = peter("first", model = fake)
branch = gptr_fork(session)
branch |> peter("branch")
c(session$turns, branch$turns)
```

gptr_hook *Define an event hook*

Description

A hook runs `handler(event, ctx)` for one event of the catalog (see `gptr_api()$features`) or for a plugin channel named `"<plugin>:<topic>"`. What the handler may return depends on the event: patches for `tool_result`, a decision for `tool_call`, nothing for notifications. Handlers of `tool_call`, `permission_request` and `document_write` fail closed: an error blocks or denies, and a `tool_call` answer with an unknown `decision` (anything but `"block"`, `"modify"` with an `input` list, or `"allow"`; for example `"deny"`) blocks.

Usage

```
gptr_hook(event, handler, matcher = NULL)
```

Arguments

`event`	Event name.
`handler`	`function(event, ctx)`.
`matcher`	`NULL`, a tool-name glob such as `"mcp__*"`, or `function(event)` returning `TRUE` for events the handler wants.

Value

A spec of class `c("gptr_hook", "gptr_spec")`, named after its event.

Examples

```
gptr_hook("tool_result", function(event, ctx) NULL, matcher = "r")
```

gptr_init *Create the project workspace* `.gptr/`

Description

Creates `.gptr/` with `settings.json`, `skills/`, `agents/`, `prompts/`, and (by default) the project-instructions file `vignette.Rmd` and `.gitignore`. Existing files are never overwritten, so calling it again is safe. There is no default path: without `path` gptr asks `Create .gptr/ in <project>? [y/N]` when you can answer, and signals an error otherwise. Creating the workspace is consent to write there; it is not trust: gptr then asks separately whether to trust the project, unless it is trusted already (see `gptr_trust()`). In a package source it offers the `^\.gptr$` line for `.Rbuildignore` and never writes it silently.

Usage

```
gptr_init(path, instructions = TRUE, gitignore = TRUE)
```

Arguments

path	The project directory (it must exist).
instructions	Write vignette.Rmd from the template when it is absent.
gitignore	Write .gptr/.gitignore from the template when it is absent.

Value

The absolute path of .gptr/, invisibly (NULL when you declined).

Examples

```
d = tempfile("proj")
dir.create(d)
gptr_init(d)
list.files(file.path(d, ".gptr"), all.files = TRUE)
unlink(d, recursive = TRUE)
```

---

gptr_jobs *List or stop gptr's background jobs and child processes*

Description

Lists the job table of this R process: background sessions, peter$bg() jobs, Shiny artifacts, the MCP server started by gptr_mcp_serve(), sub-agent workers and subscription-CLI children. A finished job whose stop was requested reads stopped or aborted, never error. Unavailable status information is shown as unknown.

Usage

```
gptr_jobs(kill = FALSE)
```

Arguments

kill	TRUE stops every job (sessions are canceled, processes are killed together with their process tree) and returns the table of what was stopped, invisibly.

Value

A gptr_jobs data frame with the columns id, kind (session, bg, artifact, mcp_serve, worker, cli), name, pid, status and started.

Options

Options of the transport and process layer (?gptr_options collects every option):

- gptr.max_active (8): concurrent HTTP transfers (global).
- gptr.connect_timeout (20), gptr.first_byte_timeout (120), gptr.idle_timeout (90): seconds; there is no total timeout on streams.
- gptr.max_retry_delay (60): seconds; a longer retry-after fails fast.
- gptr.max_attempts (4): transport attempts per request.
- gptr.wire_log (FALSE): TRUE writes one redacted JSON line per request start and end to <workspace root>/cache/tmp/wire-<session id>.jsonl; a path must lie inside the workspace root or tempdir().
- gptr.supervise (NULL): processx supervision of children; NULL means on, except under R CMD check.
- gptr.stdin_timeout (60): seconds a child may take to accept pending stdin bytes.

Examples

```
gptr_jobs()
```

---

gptr_login *Sign in to a model provider or an MCP server*

Description

Stores a credential for a model provider (for example "openrouter") or for an MCP server ("mcp:<server>"). With method = "auto", providers and MCP servers that support OAuth get the authorization-code flow with PKCE (S256): a page on 127.0.0.1 receives the redirect when 'httpuv' and 'later' are installed; otherwise you paste the address your browser ends on. Everything else gets masked key entry. Credentials go to the credential store (auth.json, readable only by you: mode 0600 on Unix, and on Windows the access control list of your user profile, so keep tools::R_user_dir("gptr", "config") inside it; or a keyring reference); access tokens stay in memory. A tool call never opens a browser: when an MCP server needs a sign-in, the call fails with an error naming this function.

Usage

```
gptr_login(provider, method = c("auto", "oauth", "key"))

gptr_logout(provider)
```

Arguments

provider	A provider id such as "openrouter", or "mcp:<server>" for an MCP server.
method	"auto" (default), "oauth" (the browser flow only) or "key" (key entry).

Value

gptr_login() returns TRUE invisibly; gptr_logout() removes the stored credential and forgets the access token held in memory, and returns TRUE invisibly when it removed something, else FALSE.

Examples

```
if (interactive()) {
  gptr_login("openrouter")
  gptr_logout("openrouter")
}
```

---

gptr_mcp *List MCP servers and their tools*

Description

Without tools, lists gptr's servers (the user mcp.json and, in a trusted project, .gptr/mcp.json) together with the servers configured for Claude Code, Claude Desktop, Codex, Cursor, VS Code and Pi, which gptr reads but never edits. Nothing is started: the era and the tool counts come from gptr's caches. With tools = TRUE, lists the tools of server (or of every usable server) with their R signatures and catalog cost, from the cache when it is fresh and otherwise by connecting (which starts stdio servers). Servers of an untrusted project are listed but never started.

Usage

```
gptr_mcp(server = NULL, tools = FALSE, refresh = FALSE)
```

Arguments

server	A server name, or NULL for all.
tools	TRUE to list tools instead of servers.
refresh	TRUE to reconnect and refresh the caches.

Value

A gptr_mcp_servers data frame (name, source, transport, era, status, tools, exposure, tokens, trusted) or, with tools = TRUE, a data frame with server, tool, signature, exposure and tokens. trusted is TRUE only for a server in your user mcp.json marked "trusted": true, whose readOnlyHint then lowers tool risk; a server of a project you have not trusted has status "untrusted".

See Also

gptr_mcp_add, gptr_mcp_remove, gptr_login, gptr_mcp_serve; vignette("mcp", package = "gptr") for setup, authentication and tool calls.

Examples

```
gptr_mcp()
```

gptr_mcp_add *Add or remove an MCP server in gptr's configuration*

Description

gptr_mcp_add() writes a server into gptr's own mcp.json: the user file under tools::R_user_dir("gptr", "config"), or .gptr/mcp.json of the project (used only when you trust the project). It never edits another harness's files. Give exactly one of command (a server started as a local process speaking stdio) and url (a Streamable HTTP server). Secret values in env and headers must be ${VAR} placeholders: they are expanded from the environment when gptr connects and never stored. Nothing connects until a tool is used.

Usage

```
gptr_mcp_add(
  name,
  command = NULL,
  args = character(),
  url = NULL,
  env = NULL,
  headers = NULL,
  exposure = "r",
  timeout = 60,
  scope = c("user", "project")
)

gptr_mcp_remove(name, scope = c("user", "project"))
```

Arguments

name	Server name: 1-64 letters, digits, _ or -. Its tools are reached as petermcp<name>$<tool>() inside the r tool.
command, args	The program and its arguments for a stdio server.
url	The endpoint of a Streamable HTTP server.
env, headers	Named character vectors: environment variables of a stdio server, HTTP headers of an HTTP server. Write secrets as ${VAR} placeholders.
exposure	How the model reaches the tools: "r" (R functions listed in the prompt), "direct" (declared as tools), "deferred" (found through peter$search()) or "hidden".
timeout	Seconds per request; progress notifications from the server extend it.
scope	"user" (default) or "project".

Value

gptr_mcp_add() returns the server spec invisibly; gptr_mcp_remove() returns TRUE invisibly when a server was removed, else FALSE.

Examples

```
proj = tempfile("proj")
dir.create(proj)
old = options(gptr.project_root = proj)
gptr_init(proj)
gptr_mcp_add("fs", command = "npx",
             args = c("-y", "@modelcontextprotocol/server-filesystem", "."),
             scope = "project")
gptr_mcp_remove("fs", scope = "project")
options(old)
unlink(proj, recursive = TRUE)
```

---

gptr_mcp_serve *Serve the live R session to other agents over MCP*

Description

Starts an MCP server inside this R session that offers `r`, `read`, `edit` and `write` to agents such as Codex, Claude Code, Claude Desktop or Cursor. It listens only on 127.0.0.1, requires a 192-bit bearer token, rejects foreign browser origins, and runs every call through your permission settings; R code evaluates in `envir`, where its objects persist. The calls belong to a dedicated gptr session whose permission mode is the one in effect when the server starts (stop and start the server to change it); like any new session, it becomes `gptr_last()`. Requests are answered while R waits at the console or inside a gptr call; one that needs your approval is refused with a note on how to allow it, because gptr never asks from a background callback. The token appears only in the snippets of `$config` (printed redacted; `as.character()` gives the text to paste; `$config$codex` holds the `codex` options in `$args` and the environment to start Codex with in `$env`). One server runs per R process. Needs 'httpuv', 'later' and 'openssl'; gptr's own codex route does not use this server.

Usage

```
gptr_mcp_serve(
  tools = c("r", "read", "edit", "write"),
  envir = parent.frame(),
  port = NULL,
  stop = FALSE
)
```

Arguments

`tools`	The tools to serve: a subset of `c("r", "read", "edit", "write")`.
`envir`	Where served R code evaluates and its objects persist.
`port`	A port, or `NULL` for a free one.
`stop`	`TRUE` to stop the server.

Details

If shutdown cannot be confirmed, access is revoked and the server is retained for retry. Call `gptr_mcp_serve(stop = TRUE)` again; new requests are refused until shutdown completes.

Value

A `gptr_mcp_handle` with `$url`, `$port`, `$token_env`, `$config` and `$stop()`; calling again returns the running handle; with `stop = TRUE`, `invisible(NULL)`.

Examples

```
if (interactive() && rlang::is_installed(c("httpuv", "later", "openssl"))) {
  h = gptr_mcp_serve(envir = new.env())
  h$config$codex
  gptr_mcp_serve(stop = TRUE)
}
```

---

gptr_models *List models from the model catalog*

Description

Searches the merged model catalog: the shipped snapshot (models.dev plus gptr's reviewed prices and capabilities), a refreshed copy in `tools::R_user_dir("gptr", "cache")`, models declared by registered providers, user configuration and discovered local servers. Offline by default: only `refresh = TRUE` touches the network.

Usage

```
gptr_models(query = NULL, provider = NULL, refresh = FALSE)
```

Arguments

`query`	`NULL` or a regular expression or alias (for example `"sonnet"`), matched against the reference, the name and the aliases; the model an alias resolves to is listed first.
`provider`	`NULL` or a provider id (for example `"anthropic"`).
`refresh`	`TRUE` downloads the current models.dev catalog with ETag revalidation into the user cache. For a local provider (`provider = "ollama"`, `"lmstudio"`, ...) it asks that server for its installed models instead (a 1 second limit per request; for Ollama the native `/api/version`, `/api/tags` and `/api/show` endpoints). Ollama discovery requires a loopback address under the default local-only policy; only an explicit human setting at user or session scope can relax it (see `gptr_config()`). The only network use of this function; it never starts or installs a server and never downloads a model.

Details

A catalog entry describes a route, not current account access. Copy its `ref` into the `model` argument of `peter()` to choose that route. Ollama routes require current installed model evidence before inference; discovery does not pull models. See `vignette("language-models", package = "gptr")` for setup and `gptr_providers()` for credentials and reachability.

Value

A `gptr_models` data frame with columns `ref`, `provider`, `name`, `context`, `max_output`, `input_price`, `output_price` (USD per million tokens, selected for today's date from the available catalog; `0` for verified local inference, `NA` when unknown), `reasoning`, `aliases` and `status`. Catalog prices can be stale and are not an account quote.

Examples

```
gptr_models("sonnet")
gptr_models(provider = "anthropic")
```

---

gptr_on *Listen to a session's events*

Description

Registers a session-scoped hook (rank 0) for one cataloged event (such as `message_end`, `tool_call`, `turn_end`) or a plugin channel containing `:`. The handler is `function(event, ctx)` and returns what the event allows. Forks never copy listeners.

Usage

```
gptr_on(s, event, handler, matcher = NULL)
```

Arguments

`s`	A session.
`event`	An event name; Claude and Codex hook names are refused with the gptr name.
`handler`	`function(event, ctx)`.
`matcher`	`NULL`, a tool-name glob for tool events, or `function(event)` returning a flag.

Value

A function of no arguments that removes the hook, invisibly.

Examples

```
session = peter("hi", model = gptr_fake_provider(list("hello")), .run = FALSE)
seen = new.env()
seen$roles = character()
off = gptr_on(session, "message_end", function(event, ctx) {
  seen$roles = c(seen$roles, event$message$role)
  NULL
})
gptr_step(session)
off()
seen$roles
```

`gptr_options` *Options that control gptr*

Description

gptr reads each option with `getOption()` at the moment it needs it, so options set with `options()` take effect at the next call. Options marked *settings* have no default of their own: when unset, the value comes from the settings layers written by `gptr_config()` (session, project `.gptr/settings.json`, user settings). The safety options (`gptr.ui`, `gptr.interactive`, the guards, `gptr.noninteractive_ask`, `gptr.protect_size`, `gptr.mode` and `gptr.unsafe_no_permissions`) are copied when a run starts, so code that the model runs cannot change them for that run.

Interaction and output

- `gptr.quiet` (logical, `FALSE`): silence notices and progress messages.
- `gptr.interactive` (logical or `NULL`, `NULL`): force the decision whether a person is present to answer questions and permission prompts. This does not supply a UI or an answer in an unattended process.
- `gptr.verbose` (integer or `NULL`, `NULL`): 0 silent, 1 progress on stderr (stdout in an ordinary IRkernel cell), 2 the streamed console, 3 debugging; `NULL` chooses by context (0 in knitr and testthat, 1 under Rscript, 2 at the console).
- `gptr.ui` (character, UI spec or `NULL`, `NULL`): the user-interface backend; `NULL` means the console when a person is present, else none.
- `gptr.history` (logical, `TRUE`): add console inputs to the R history.
- `gptr.deprecations` (character, `"warn"`): `"warn"` or `"error"` for deprecated calls.

Settings, models and replay

- `gptr.model`, `gptr.mode`, `gptr.preset`, `gptr.system1`, `gptr.small_model` (character or `NULL`, *settings*): the option layer of the settings of the same names.
- `gptr.project_root` (character or `NULL`, `NULL`): the project root; also the environment variable `GPTR_PROJECT_ROOT`.
- `gptr.replay` (character or `NULL`, *settings*, `"auto"`): how recorded document blocks are used: `"auto"`, `"replay"`, `"live"` or `"record"`; the `replay` argument of `peter()` wins.

- gptr.record (character or NULL, *settings*, "ask"): whether gptr may write into documents: "auto", "ask" or "off".
- gptr.interpolate (logical, TRUE): {identifier} interpolation in literal prompts.

Sessions, turns and values

- gptr.max_turns (integer, 50): turns per programmatic run.
- gptr.max_turns_console (integer, 200): turns per console prompt.
- gptr.max_nested_calls (integer, 20): peter() calls per evaluation of model code.
- gptr.value_copy_max (bytes, 2097152): designated result values below this size are copied; larger ones are kept by name.
- gptr.values_max_bytes (bytes, 67108864): value copies held per session.
- gptr.background_tools (character, "idle"): "idle" or "wait" for experimental background sessions.
- gptr.out_keep (integer, 20): results kept per session for peter$out().
- gptr.spill_days (number, 7): age in days after which spill files are pruned.

Concurrency and sub-agents

- gptr.max_active (integer, 8): concurrent HTTP transfers.
- gptr.subagents.max_active (integer, 8): concurrent inline sub-agents; the default of gptr_parallel()'s max_active.
- gptr.subagents.max_cli (integer, 4): concurrent command-line sub-agents.
- gptr.subagents.max_workers (integer or NULL, NULL): worker processes; NULL means min(4, cores - 1). Under R CMD check every pool of child processes is capped at 2.
- gptr.subagents.max_tasks (integer, 8): children per team or fan-out started by model code.
- gptr.subagents.max_depth (integer, 1): nesting depth of child sessions (at most 2).
- gptr.child_text_max (bytes, 51200): text returned per child task.

Transport and processes

- gptr.connect_timeout (seconds, 20), gptr.first_byte_timeout (seconds, 120) and gptr.idle_timeout (seconds, 90): network timeouts. High-effort reasoning on OpenAI or Google may need options(gptr.idle_timeout = 300).
- gptr.max_retry_delay (seconds, 60): a longer retry-after from a provider fails at once.
- gptr.max_attempts (integer, 4): transport attempts per request.
- gptr.wire_log (logical or path, FALSE): write a redacted log of every request and response inside the workspace or the session temporary directory.
- gptr.supervise (logical or NULL, NULL): supervise child processes; NULL means yes except under R CMD check.
- gptr.stdin_timeout (seconds, 60): time allowed to write pending input to a child.
- gptr.cli_path (named list or NULL, NULL): explicit paths of the claude and codex command-line tools.
- gptr.cli_turn_timeout (seconds, 3600): wall-clock time per command-line turn.

Evaluation and tools

- `gptr.r_timeout` (seconds, `3600`): time limit of the `r` tool when nobody is present. It stops R code that is computing; in a non-interactive R it does not end `Sys.sleep()` or `system()`, which `peter$sh(timeout = )` does for a command.
- `gptr.r_output_tokens` (integer, `4000`): estimated tokens of one `r` result.
- `gptr.r_max_images` (integer, `3`): plot images attached to one `r` result.
- `gptr.helper_output_tokens` (integer, `1500`): printed size of `peter$` helper results.
- `gptr.read_max_tokens` (integer, `12000`): size cap of the `read` tool.
- `gptr.plot_width`, `gptr.plot_height`, `gptr.plot_res` (integers, `768`, `512`, `120`): plots sent to the model.

Permissions

- `gptr.noninteractive_ask` (character, `"stop"`): what a permission question does when nobody can answer: `"stop"` ends the run with status `blocked` and a `gptr_error_permission` condition (`gptr_error_noninteractive` when the `ask` tool has no one to answer); `"deny"` returns a denial to the model.
- `gptr.critical_guard` (logical, `TRUE`): level-4 actions ask even in `auto` mode.
- `gptr.secret_guard` (logical, `TRUE`): reads of registered secrets ask even in `auto` mode; gptr_security lists the reads.
- `gptr.protect_size` (bytes, `1e8`): overwriting a larger object is a level-3 action.
- `gptr.plan_handoff` (logical, `TRUE`): hand the plan a `plan` run proposes, once, to the next top-level `peter()` call from the same environment, within an hour and in another mode; another `peter()` call first, or a next call inside a loop or a run, discards it.
- `gptr.unsafe_no_permissions` (logical, `FALSE`): turns the permission gate off entirely; honored only when set outside a run, for sandboxed continuous integration.

Context, caching and compaction

- `gptr.compact_at` (tokens or `NULL`, `200000`): the compaction soft cap; `NULL` disables the cap.
- `gptr.compact_cold_min` (tokens, `100000`): the size above which a cold cache compacts.
- `gptr.cache_ttl` (character, `"gap"`): prompt-cache lifetime policy: `"gap"`, `"5m"` or `"1h"`.
- `gptr.cache_gap` (seconds, `240`): a pause between requests longer than this switches the cache lifetime to one hour.
- `gptr.check_prefix` (character, `"event"`): what a broken cache prefix does: `"event"`, `"warn"` or `"error"`.
- `gptr.skills_budget` (integer, `1500`) and `gptr.mcp_budget` (integer, `1500`): tokens of the skill and MCP tool catalogs.

Checkpoints and undo

- `gptr.checkpoint` (character, `"on"`): `"on"`, `"files"` or `"off"`.
- `gptr.undo_capture_max` (bytes, `1e8`): objects up to this size are captured by reference before a change, even when the change was not predicted.

- gptr.undo_max_bytes (bytes, 1e9): object images held in memory per session.
- gptr.undo_spill_max (bytes, 2e9): the largest object image written to disk.
- gptr.undo_turns (integer, 20): object images older than this many turns are dropped.
- gptr.checkpoint_disk_bytes (bytes, 2e9), gptr.checkpoint_days (days, 30) and gptr.checkpoint_turns (integer, 100): retention of the checkpoint store.
- gptr.checkpoint_track_file_max (bytes, 1e6) and gptr.checkpoint_track_total (bytes, 1e8): the per-file and total size of tracked project files.
- gptr.checkpoint_capture_max (bytes, 5e7): a larger changed file gets no stored copy.
- gptr.checkpoint_scan_budget (seconds, 0.25): above this walk time the project is scanned once per turn.
- gptr.checkpoint_rng (logical, TRUE): report changes of the random-number state in the rewind report (gptr never changes your random seed).
- gptr.checkpoint_close_devices (logical, FALSE): close graphics devices opened by an undone turn.

Secrets

- gptr.redact_min_chars (integer, 8): the shortest secret value that is redacted.
- gptr.redact_patterns (logical, TRUE): also redact text that looks like a key; known values are always redacted.
- gptr.stream_hold_max (integer, 4096): characters held back while redacting a stream.
- gptr.env_export (logical, TRUE): the default of gptr_env(set_env =).
- gptr.prompt_secrets (character, "redact"): secret-looking text in prompts: "redact" or "ask".

System 1

- gptr.s1_max_active (integer, 8): concurrent System 1 requests.
- gptr.s1_rounds (integer, 3): retry rounds for failed elements.
- gptr.s1_state_max (integer, 2000): characters of a session's state sent to System 1.
- gptr.s1_max_elements (integer, 10000): elements per System 1 call.

Documents, MCP and artifacts

- gptr.doc_output_lines (integer, 12): output lines recorded per execution.
- gptr.doc_source_frames (logical, TRUE): let gptr find the calling line of a script run with source().
- gptr.mcp_timeout (seconds, 60) and gptr.mcp_probe_timeout (seconds, 5): MCP request and protocol-probe time limits.
- gptr.mcp_debug (logical, FALSE): keep redacted MCP server logs in the user cache.
- gptr.artifact_max_bytes (bytes, 5e8): the largest artifact data snapshot.

Environment variables

- GPTR_REPLAY: the replay mode below the gptr.replay option; GPTR_REPLAY=replay proves that a script makes no model calls.
- GPTR_PROJECT_ROOT: overrides the project root, like gptr.project_root.
- ANTHROPIC_API_KEY, OPENAI_API_KEY, GEMINI_API_KEY, TYPESAFE_API_KEY and the other provider keys: read at the first use of a provider; see gptr_env() for .env files.
- R_USER_CONFIG_DIR, XDG_CONFIG_HOME and HOME (on Windows also APPDATA and USERPROFILE): they place tools::R_user_dir("gptr", "config"), which holds gptr's settings, permission rules, trust decisions and credentials, and your home directory. Setting or unsetting one of them from model code changes gptr's own configuration, as does a variable a registered provider reads (its key or base-URL variable, such as AWS_REGION for Bedrock, or a variable gptr keeps for the claude or codex command-line tool: its login, its folder or the path of Git Bash, such as CODEX_HOME), a GPTR_* or *_BASE_URL variable, R_ENVIRON_USER, R_PROFILE_USER, R_ENVIRON or R_PROFILE, whatever the letter case. Every mode but yolo asks you first, auto included, before Sys.setenv(), Sys.unsetenv(), withr::local_envvar() or withr::with_envvar() of such a name or of a name the classifier cannot read, before readRenviron() and dotenv::load_dot_env(), and before an os.environ write in peter$py() or peter$knit("python", code) code that names one in a string or an upper-case word or whose name is not a plain string literal; gptr_security lists what it does not read.
- GPTR_WORKER, GPTR_SUBAGENT_DEPTH, GPTR_ARTIFACT_PORTS, GPTR_SHELL_CMD and the tree markers GPTR_PROC_<id>: set by gptr in its own child processes only.
- GPTR_MCP_TOKEN: the variable through which the Codex snippet of gptr_mcp_serve() passes the server's bearer token to the client; gptr itself never reads it.
- GPTR_LIVE_TESTS: "true" enables the package's live tests; not used at run time.
- _R_CHECK_PACKAGE_NAME_: set by R CMD check; outside the package's own tests a model call without its own replay argument then runs in replay mode (recorded answers or an offline provider, never a model server), and each child-process pool is capped at two.

See Also

gptr_config() for settings stored in files, gptr_security and gptr_egress; vignette("configuration", package = "gptr") for scope, precedence and grouped examples.

---

gptr_parallel *Run several agent calls concurrently*

Description

Each argument is a peter() call; it is evaluated with deferral on, so it builds its session without running it, and all members then run together on one reactor, at most max_active at a time. The result is a team session: team$text joins the members' answers under ### <name> (<model>) headings (a failed member's says (ended with status error)), team$value is the named list of their values, and team$<name> (or team[[i]]) is a member's session. Piping the team into peter() continues it: then team$text (and printing) is the team's own answer, team$value its own value once a turn designates one, and each member's answer stays at team$<name>$text.

Usage

```
gptr_parallel(
  ...,
  .list = NULL,
  max_active = NULL,
  on_error = c("return", "stop")
)
```

Arguments

`...`	Named `peter()` calls (or sessions created with `.run = FALSE`).
`.list`	A named list of sessions created with `.run = FALSE`.
`max_active`	The number of members that run at once; `NULL` uses the option `gptr.subagents.max_active` (8).
`on_error`	`"return"` keeps failed members with their status; `"stop"` signals the first failed member's condition once every member settled.

Details

Members evaluate R code in the environment their `envir =` names, with their own random-number stream, and share one budget, as one `peter()` call does. A member without `envir =` runs in the caller's environment, which it shares with the other members, so members that assign the same name overwrite each other: give each its own `envir = new.env()`. A member's `tools =`, `budget =` and the run options in its `.opts` (such as `max_turns`) are not applied.

Use `peter(..., agents = list(...))` for named roles on one task, or `parallel =` for one task per input element. Those routes start new foreground children, cannot be deferred as members here, and make a recorded team; the members of `gptr_parallel()` are nested calls that record nothing, so a history document (see `gptr_doc()`) that holds one cannot replay or export it. A team report is model-generated output; inspect child statuses and workspace results before continuing the team with a synthesis prompt.

Value

A team session (`kind = "team"`).

Examples

```
fake = gptr_fake_provider(list("ok"))
session = gptr_parallel(planner = peter("Plan it", model = fake, envir = new.env()),
                        review = peter("Review it", model = fake, envir = new.env()))
names(session$children)
```

gptr_permissions *Manage permission rules*

Description

Lists the permission rules in effect, or adds and removes rules. A rule is `tool` or `tool(spec)`: a path glob for `read`, `write`, `edit`, `grep`, `find` and `ls` (`write(results/**)`, relative to the project root; `~/...` for the home; `//...` for other absolute paths); for `r`, `level<=n`, `fn:name,...` (every flagged function covered), `category:name,...`, `sh:<command glob>` (`r(sh:git push*)`; a glob ending in `*` also matches the command alone, so `r(sh:cat *)` covers `cat` too; a glob matches the command as written, in any case on macOS and Windows, so it misses options placed before a subcommand, as in `git -C dir push`), `sql:<keywords>` (`r(sql:select)`) and `secret:NAME` (the only rule that pre-approves a guarded secret read: an allow rule covers `r` code only when every call it flags reads one of the named secrets by name, with `Sys.getenv()` or a shell's `$NAME`; it covers a secret handed to a child through the `env` of `peter$sh()` or `peter$script()` only when the same code reads that secret by name, while a deny or ask `secret:NAME` rule meets every such hand-over); MCP tools by name (`mcp__github__*`). A tool name names a tool the model calls, never a `peter$` member, served or not: `r(fn:peter$script)` names `peter$script()` in `r` code. Deny rules win over ask rules, which win over allow rules. Deny and ask rules match any call that `gptr_risk()` flags, a read-only shell command included (`r(sh:cat *)` denies `peter$sh("cat notes.txt")`); an R function it knows to be read-only, such as `readLines()`, is not flagged, so `fn:` rules never match it. Allow rules never loosen plan mode and never pre-approve level-4 (critical or control) actions. In `auto` and `yolo` mode only an `fn:` rule that names the function, such as `r(fn:system)`, lets `system()`, `system2()`, `shell()`, `shell.exec()`, `pipe()`, `processx::run()` (`r(fn:run)`) or `data.table::fread(cmd =)` run a command the code computes, which those modes otherwise refuse (see gptr_security).

Usage

```
gptr_permissions(
  allow = NULL,
  ask = NULL,
  deny = NULL,
  remove = NULL,
  scope = c("session", "project", "user")
)
```

Arguments

`allow`, `ask`, `deny`	Character vectors of rules to add to that list, or `NULL`.
`remove`	Character vector of rules to remove from every list of `scope`, or `NULL`.
`scope`	One of `"session"`, `"project"`, `"user"`.

Details

Scopes: `"session"` is this R process; `"project"` is the user-level file 'projects/<hash>.json' in `tools::R_user_dir("gptr", "config")`, personal and never inside the project tree; `"user"` is

the user settings file. The listing also shows the shared project rules of `.gptr/settings.json` (an untrusted project contributes only `deny` and `ask` rules) and the `deny`/`ask` rules of a legacy `.gptr/settings.local.json` in a trusted project.

Called from model code during a run, adding or removing rules needs the user's approval of that call; otherwise it signals `gptr_error_permission`.

Value

A `gptr_permissions` data frame with columns `rule`, `list`, `scope` and `source`; invisibly when rules were added or removed.

Examples

```
gptr_permissions(allow = "r(level<=1)")
gptr_permissions()
gptr_permissions(remove = "r(level<=1)")
```

`gptr_plugins` *Plugins known to this session*

Description

Lists the plugins gptr can see: plugins enabled through `peter(plugins = )` or the `plugins` setting (a settings plugin not enabled yet shows as `lazy`), attached packages that ship `inst/gptr/`, the project's `.gptr/plugins/` directories and installed Claude Code plugins. With `installed = TRUE` it also scans every installed package for `inst/gptr/`. Nothing is loaded or run. A plugin enabled both for the process and for a live session is listed once.

Usage

```
gptr_plugins(installed = FALSE)
```

Arguments

`installed`	`TRUE` to scan every installed package as well.

Details

Enable a plugin with `peter(plugins = "name")` or the `plugins` configuration setting. Packages put resources in `inst/gptr/`; directory plugins use a root `plugin.json`. R factories activate lazily only when the manifest declares `extension.provides` and does not request eager activation. Claude bundles contribute declarative resources; their hooks and executable plugin behavior are not imported.

Value

A `gptr_plugins` data frame with columns `name`, `version`, `api`, `kind` (`package`, `directory`, `claude-plugin`), `enabled`, `state` (`lazy`, `active`, `disabled`, `failed`), `provides`, `tokens` and `path`.

See Also

gptr_check, gptr_reload, gptr_registry; `vignette("skills-and-plugins", package = "gptr")` for resource files and plugin manifests.

Examples

```
gptr_plugins(installed = TRUE)
```

`gptr_policy` *Define a permission policy*

Description

`check(call, ctx)` sees every tool call before it runs and returns `NULL` (no opinion) or `list(decision = "allow" | "deny" | "ask" | "ask_human" | "modify", reason, input)`. Decisions combine as deny > ask_human > ask > modify > allow; only a person answers an `ask_human`, never a hook; a policy that errors denies. Keep checks under 10 ms.

Usage

```
gptr_policy(name, check, description = NULL)
```

Arguments

`name`	Policy name.
`check`	`function(call, ctx)`.
`description`	One line for listings.

Value

A spec of class `c("gptr_policy", "gptr_spec")`.

Examples

```
gptr_policy("no_installs", check = function(call, ctx) {
 if (identical(call$name, "r") && grepl("install.packages", call$input$code, fixed = TRUE)) {
    list(decision = "deny", reason = "installs are not allowed here")
  } else {
    NULL
  }
})
```

gptr_preimage *Tell the object checkpointer how to capture an object*

Description

gptr_preimage() is the S3 generic the object checkpointer calls for a binding of the workspace before a tool call that runs R code. It decides how the value the binding holds now is kept, so that a rewind can put it back. Methods are leaf functions: they may read x but must never store it anywhere.

Usage

```
gptr_preimage(x, name, ctx, ...)

## Default S3 method:
gptr_preimage(x, name, ctx, ...)

## S3 method for class 'data.table'
gptr_preimage(x, name, ctx, ...)
```

Arguments

x	The object bound to name.
name	The binding name (a string).
ctx	A list: predicted, what the agent's code is predicted to do to name ("assign", "modify", "byref", "remove" or "none"); bytes, the size of x in bytes (NA when unknown); budget, the largest copy allowed, in bytes. When ctx$dry is TRUE the checkpointer only asks whether a capture is possible and a method must not copy.
...	Unused; for methods.

Details

The default method keeps value-semantics objects by reference ("ref": nothing is copied unless the agent edits the object in place) and reports environments (including R6 and RefClass objects) and external pointers as not restorable ("none"). The data.table method returns a deep copy made with data.table::copy() ("copy") when the agent's code is predicted to modify the table by reference (:=, set*()) and the table fits the budget. Packages add methods for their own classes with a delayed registration, S3method(gptr::gptr_preimage, myclass); an R6 class can, for example, return x$clone(deep = TRUE) as "copy". This generic is experimental.

Value

A list with mode ("ref", "copy" or "none"), reason (a string explaining "none", else NULL) and copy (the deep copy for "copy", else NULL).

Examples

```
gptr_preimage(1:3, "x", list(predicted = "assign", bytes = 64, budget = 1e9))$mode
gptr_preimage(new.env(), "cfg", list(predicted = "modify", bytes = NA, budget = 1e9))$reason
```

---

gptr_prob *Probabilities and confidence of System 1 answers*

Description

System 1 calls (`peter(question, x, model = jev)`) return typed vectors: a `gptr_decision` (logical), a `gptr_choice` (character) or a `gptr_score` (double, the expected 0-based level). They work in `if()`, `while()`, `ifelse()`, `table()`, `sum()` and comparisons like the bare vectors, and carry the model's probabilities as attributes. `gptr_prob()` reads them. Choices given as a factor, or `.opts = list(output = "factor")`, return a plain factor instead, which carries no probabilities.

Usage

```
gptr_prob(x, what = c("prob", "confidence", "probabilities"))
```

Arguments

`x`	A System 1 vector: a `gptr_decision`, `gptr_choice` or `gptr_score`.
`what`	`"prob"` (decisions: P(yes); choices: the probability of the chosen option; scores: the confidence), `"confidence"` (decisions: `abs(2 * p - 1)`) or `"probabilities"` (a matrix with one row per element and one column per option or level; for decisions the columns `FALSE` and `TRUE`).

Details

Probabilities are the model's own; they are empirically calibrated only when the printed footer says `calibrated`. Native decision models report calibration as unknown unless calibration evidence was recorded, and emulated answers are always `uncalibrated`. Answers combined from several calls (`c()`, `[<-`, `vctrs::vec_c()`) say `calibrated` only when every part is calibrated and `uncalibrated` when any part is uncalibrated.

Confidence is not the probability that the answer is correct. For a decision it is `abs(2 * p - 1)` on every engine, so `min_confidence = 0.8` keeps only P(yes) at most 0.1 or at least 0.9. For choices and scores it is the engine's own measure of how concentrated the distribution is. TypeSafe Jev (and emulation) reports `(n * max(p) - 1) / (n - 1)` for a choice and `1 - E|level - mode| / d` for a score, `d` being the levels' mean distance from their middle; Ollama's Clef and Clef Flash report `1 - H(p) / log(n)` for both. The distribution 0.90, 0.05 and 0.05 has confidence 0.85 on Jev and 0.64 on Clef, so re-tune `min_confidence` when switching engine.

A score can be fractional: with levels low, medium and high (0, 1 and 2), probabilities 0.85, 0.10 and 0.05 give the score 0.20. It is not rounded to the most likely level. `attr(x, "meta")` records the model, engine, calibration state, cached elements, usage and available execution provenance such as locality, model digest and server version.

Value

A named numeric vector, or a matrix for `what = "probabilities"`.

Input shape and uncertainty

Atomic vectors and unnamed lists give one input state per element; data frames give one per row. Fully named lists and `I(x)` give one structured state. These states can contain the actual input values sent to the provider; see gptr_egress. `min_confidence` makes low-confidence answers `NA` by default, including unknown confidence when the minimum is positive. Use `isTRUE()` in scalar control flow when abstention is possible, and handle uncertain inputs separately.

System 1 options

`gptr.s1_max_active` (8) concurrent requests, `gptr.s1_rounds` (3) bounded retry rounds, `gptr.s1_state_max` (2000) characters of a piped session's state and `gptr.s1_max_elements` (10000) elements per call.

Examples

```
judge = gptr_fake_provider(list(0.9, 0.2), name = "judge", type = "classifier")
d = peter("Is this about dogs?", c(a = "A puppy.", b = "A car."), model = judge)
gptr_prob(d)
```

---

gptr_prompt *Show the frozen system prompt, tool array and first message*

Description

`gptr_prompt()` shows what gptr sends before the conversation: the two system blocks (T0, the static sections; T1, the machine and project sections), the tool array and the first user message, with an estimated token count per section. It makes no model call and writes nothing: a preview never consumes a pending plan and queues no message for the model.

Usage

```
gptr_prompt(x = NULL, preset = NULL, tokens = TRUE)

## S3 method for class 'gptr_prompt_view'
print(x, ...)
```

Arguments

`x`	For `gptr_prompt()`, a `gptr_session` (its frozen prompt and first message) or `NULL` (what a new session would freeze now with the current settings); for `print()`, a `gptr_prompt_view`.
`preset`	`NULL` or a preset name (`"minimal"`, `"standard"`, `"readonly"`, `"extended"`, or one registered by a plugin or for the session) to preview instead of the frozen or configured one.
`tokens`	`TRUE` to add estimated token counts per section.
`...`	Unused.

Details

For a session, the view shows its frozen blocks (restored from its `gptr.frozen` entry when needed) and the text of its first user message, context blocks included. With `preset`, for a session that has not frozen its prompt yet, or for one whose next run freezes a fresh prompt (resumed from a file of another project or one that git tracks), it composes what would be frozen now.

Value

A `gptr_prompt_view`: a list with `system` (`list(t0, t1)`), `tools_json` (the tool array as JSON text), `first_message` (text), `sections` (data frame `name`, `tier`, `tokens`) and `total_tokens`; the attributes `preset` and `tool_names` name the preset and the declared tools. `print()` shows each part with its token estimate and returns `x` invisibly.

Examples

```
v = gptr_prompt(preset = "minimal")
v$sections
```

---

gptr_prompt_section *Define a system-prompt section*

Description

A section of the frozen system prompt, rendered once per session in ascending `order`: `T0` sections form the stable prefix, `T1` sections follow. `text` may be a function of `ctx`; `parent` makes the section a fragment of another section.

Usage

```
gptr_prompt_section(
  name,
  text,
  tier = c("T0", "T1"),
  order = 500L,
  budget = 300L,
  parent = NULL
)
```

Arguments

`name`	Section name (the tag it is wrapped in).
`text`	A string or `function(ctx)` returning a string or `NULL`.
`tier`	`"T0"` or `"T1"`.
`order`	Position among sections.
`budget`	Token budget of the section.
`parent`	Name of the section this fragment belongs to, or `NULL`.

Value

A spec of class `c("gptr_prompt_section", "gptr_spec")`.

Examples

```
gptr_prompt_section("house_rules", "Use SI units in every table.", tier = "T1", order = 780L)
```

---

gptr_provider *Define a model provider*

Description

A provider is data: an id, the wire `api` its adapter implements, a base URL, how to find a credential and its models. `auth` names environment variables (or is a function returning a secret handle) and is resolved per request.

Usage

```
gptr_provider(
  id,
  api,
  base_url = NULL,
  auth = NULL,
  models = NULL,
  compat = list(),
  type = c("chat", "classifier", "cli"),
  headers = list(),
  discover = NULL,
  status = NULL,
  aliases = character(),
  local = FALSE,
  offline = FALSE,
  rate = NULL
)
```

Arguments

`id`	Provider id, `^[a-z0-9][a-z0-9-]*$`.
`api`	Wire api (the name of an adapter), for example `"openai-completions"`.
`base_url`	Endpoint base URL.
`auth`	Character vector of environment-variable names, a function, or `NULL`.
`models`	List of model records (each with at least `id`). A reasoning model (`reasoning = TRUE`) lists the levels it accepts in `thinking_levels`, a subset of `"off"`, `"minimal"`, `"low"`, `"medium"`, `"high"`, `"xhigh"` and `"max"`; without it `"off"` to `"high"` is assumed. A requested level the model lacks becomes the nearest one it has, with a message. A record's `prices` is a data frame or a list of price records: `input`

	and output in USD per million tokens, optional cache_read, cache_write_5m, cache_write_1h, from (a "YYYY-MM-DD" date) and tier ("default", or ">Nk" for prompts above N thousand tokens, such as ">200k"); it is checked here, and without it the cost is unknown.
compat	Named list of compatibility switches for the adapter (for api = "bedrock-converse", region: the AWS region requests are signed for).
type	"chat", "classifier" (System 1) or "cli".
headers	Named list of non-secret header strings.
discover	function() listing models on request, or NULL.
status	function() reporting cached provider status, or NULL.
aliases	Alternative names of the provider.
local	Endpoint locality hint permitting unknown model ids. Resolved model locality is checked separately: a loopback server may route inference to a remote model.
offline	TRUE when no remote model is called (test providers).
rate	list(requests_per_s, tokens_per_s) static limits, or NULL.

Value

A spec of class c("gptr_provider", "gptr_spec").

Examples

```
gptr_provider("corp", api = "openai-completions", base_url = "https://llm.corp.example/v1",
          auth = "CORP_LLM_KEY", models = list(list(id = "corp-large", context = 128000)))
# Amazon Bedrock's Converse API through a SigV4 gateway (region and endpoint are record data):
# the IAM key pair is read from AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY (and
# AWS_SESSION_TOKEN when set); gptr_env(aliases = ...) maps other names onto them. Thinking
# levels are sent only to a Claude model whose `capabilities` declare adaptive thinking with
# an effort (Claude 4.6 and later); other models get no thinking, with a message. The
# bedrock-converse adapter does not request prompt caching, so each request sends the full
# prompt at the uncached rate.
gptr_provider("gateway", api = "bedrock-converse", base_url = "https://ai.corp.example/awsai",
              auth = "AWS_SECRET_ACCESS_KEY", compat = list(region = "us-west-2"),
              models = list(list(id = "us.anthropic.claude-opus-4-8", tool_call = TRUE,
                                 reasoning = TRUE,
                            capabilities = list(adaptive_thinking = TRUE, effort = TRUE))))
```

---

gptr_providers *List the configured model providers*

Description

Shows every registered provider record: its wire api, where its credential comes from (as NAME #fingerprint, never the value), the registry source of the record, its status, default model, egress acknowledgment and, for subscription command-line tools, their version.

Usage

```
gptr_providers(check = FALSE, check_login = FALSE)
```

Arguments

check	FALSE (default) performs no network or process input/output unless check_login = TRUE. TRUE also probes each HTTP provider's models endpoint once without credentials (2 s timeout; skipped under R CMD check) and lets command-line providers check their tool; it never sends a paid request.
check_login	Also check Codex and Claude Code's CLI-reported login status using their supported status commands (3 s timeout per subprocess, no model request). Adds a login column: signed in, not signed in, or unknown. Unsupported CLI versions, missing tools and failed checks report unknown; other providers report not applicable. A supported API-key login can count as signed in; this does not verify credentials online or establish subscription billing. Defaults to FALSE.

Details

status is ready, no key, no base url or invalid base url (HTTP providers), disabled (the settings say providers.<id>.enabled: false), error (the provider's settings or its credential lookup failed) or what a command-line provider reports; with check = TRUE the ready or no key of an HTTP provider becomes reachable (HTTP <code>) for a reachable endpoint (a 401 without a key still proves reachability), otherwise unreachable, or not checked under R CMD check. egress is ack when no acknowledgment is needed (offline providers, local servers at a loopback address) or the user has given it. Listing reads credentials only: it never registers an environment variable or binds it to a provider.

ready means the required configuration is present; it does not validate the credential, account quota or access to a specific model. check = TRUE tests reachability, not login. See vignette("language-models", package = "gptr") for API, command-line and local setup.

Value

A gptr_providers data frame with columns id, type, api, credential, source, status, default_model, egress (ack or needed) and version, plus login when check_login = TRUE.

Examples

```
gptr_providers()
```

---

gptr_redact *Redact secrets from text or from a nested list*

Description

Replaces the values of registered secrets (and their URL-encoded, JSON-escaped and base64 forms) with markers such as `[secret:TYPESAFE_API_KEY]` and, except for the `user_data` profile, applies the pattern layer (provider key shapes, bearer tokens, URL passwords, private keys, `NAME=value` lines). gptr applies the same redactor at every sink; plugins and users call it for their own logs. Redaction is idempotent and never fails on invalid UTF-8 (it then matches bytewise). Opaque replay fields of messages (signatures, encrypted reasoning) are never modified.

Usage

```
gptr_redact(
  x,
  profile = c("persist", "stream", "context", "code", "user_data")
)
```

Arguments

`x`	A character vector, or a list (redacted recursively).
`profile`	The sink profile: `"persist"` (files), `"stream"` (console and events), `"context"` (text sent to a model), `"code"` (R code for a history document: a string literal whose value is a registered secret becomes `Sys.getenv("NAME")`) or `"user_data"` (registered values only).

Value

`x` with secrets replaced by markers; the same type and attributes as `x`.

Options

`gptr.redact_min_chars` (8): shortest value that is value-redacted. `gptr.redact_patterns` (`TRUE`): the pattern layer; values are always redacted. `gptr.stream_hold_max` (4096): streaming hold-back cap in characters. `gptr.env_export` (`TRUE`): default of `gptr_env(set_env =)`. `gptr.prompt_secrets` (`"redact"`): secret-looking text in prompts, `"redact"` or `"ask"`. `gptr.secret_guard` (`TRUE`): guarded secret reads ask even in `auto` mode.

Examples

```
gptr_redact("Authorization: Bearer abcdef0123456789abcdef")
gptr_redact(list(note = "db at postgres://analyst:hunter2x@db.example.test/prod"))
```

gptr_register *Register a capability spec*

Description

Registers a spec made by `gptr_spec()` or one of the spec constructors at top level: rank 3, source `"user"`, for the lifetime of the R process. Sessions frozen before the call are not changed; the next session sees the record. A user record overrides a built-in record of the same kind and name only (the built-in's other records keep working). Registering the same kind and name again replaces your earlier record, with a notice, and the function the earlier call returned then removes nothing (it returns `FALSE`); kinds that keep every record, such as hooks and policies, keep both.

Usage

```
gptr_register(spec)
```

Arguments

`spec`	A spec (class `gptr_spec`).

Details

Called from model code while a run executes a tool, `gptr_register()` and the function it returns signal `gptr_error_permission` unless the user approved that call.

Value

A function that removes the record again, invisibly.

Examples

```
off = gptr_register(gptr_command("hello", function(args, ctx) "hi"))
off()
```

gptr_registry *List the extension registry*

Description

One row per process-level record (session-scoped records are not listed): its kind, name, source (`builtin:<name>`, `plugin:<name>`, `user`, `project`), rank, state (`lazy`, `active`, `overridden`, `disabled`), the estimated tokens its declaration costs, and whether the kind is experimental. With `diagnostics = TRUE`, the redacted diagnostics log instead.

Usage

```
gptr_registry(kind = NULL, diagnostics = FALSE)
```

Arguments

kind	NULL for every kind, or a character vector of kind names.
diagnostics	TRUE to return the diagnostics log.

Value

A gptr_registry data frame (columns kind, name, source, rank, state, tokens, experimental), or a gptr_diagnostics data frame (columns time, source, event, class, message).

Examples

```
gptr_registry("tool")
gptr_registry(diagnostics = TRUE)
```

---

gptr_reload *Reload extensions*

Description

Bumps the registry generation, so that extension API objects captured by old code signal gptr_error_stale_api, and re-declares lazily activated plugins from their manifests (their factories run again on first use). Extension files that failed to load are loaded again at the next session start. Discovery of declarative resources (skills, prompts, agents, MCP configurations, plugin manifests) is keyed on the generation, so it runs again on next use. Sessions whose prompt is already frozen are not changed. Refused from model code while a run executes a tool, unless the user approved that call.

Usage

```
gptr_reload()
```

Value

The new registry generation (an integer), invisibly.

Examples

```
gptr_reload()
```

gptr_return *Designate the result of an agent run*

Description

Called by R code the agent runs (or by you inside a tool) to designate the run's result, which the session then returns as `$value`: the newest value designated in any turn, so a turn that designates nothing keeps it (`$values` lists them by turn; `gptr_return(NULL)` clears it). Objects bound in the session's workspace are kept by name when large (no copy), copied when small; other values are boxed. Outside a run it returns its argument invisibly and designates nothing; the first such call in an R session prints a one-line notice (silent in replay mode, `replay = "replay"` or `GPTR_REPLAY=replay`, with `options(gptr.quiet = TRUE)` or `suppressMessages()`).

Usage

```
gptr_return(x)
```

Arguments

`x`	The value. A history document records a bare name by name, and any other argument of a top-level call (also piped or over several lines) as an assignment to a hidden name in the block, which replay reads. Inside another expression (a branch, braces, an assignment or a function) such a value is not recorded, nor an earlier one it may replace, and replaying its block in a later R session says so once.

Value

`NULL` invisibly during a run; `x` invisibly outside a run.

Examples

```
y = gptr_return(1:3)
y
```

gptr_rewind *Rewind a session: undo turns, restore objects and files, move the conversation*

Description

`gptr_rewind()` is a branch-in-place on the one session object: it undoes the checkpoint records of the turns being left (newest first), redoes the records of a target on another branch (oldest first) and appends one `gptr.rewind` entry parented at the target (the session file stays append-only and the rewind survives a reload). It hands back the undone prompt in `s$editor_text` and returns `s` invisibly, so it pipes: `session |> gptr_rewind() |> peter("Try another way")`. Use `gptr_fork()` for a second session.

`gptr_checkpoints()` lists one row per turn of the active path (`all = TRUE` adds the turns of abandoned branches): `turn`; `id`, the last entry of the turn (pass it as `to` to `gptr_rewind()`); `time`, when the prompt was sent (local time); `prompt` (its first 60 characters); `objects` and `files`, `"changed/restorable"` counts; `held_mb` and `disk_mb`, megabytes (2^20 bytes) of object pre-images held in memory and on disk; and `branch` (`"active"` or `"abandoned"`).

Usage

```
gptr_rewind(
  s,
  turn = -1L,
  to = NULL,
  restore = c("all", "conversation", "workspace"),
  force = FALSE,
  preview = FALSE
)

gptr_checkpoints(s, all = FALSE)
```

Arguments

`s`	A `gptr_session`.
`turn`	Integer. Negative counts back from the last turn (`-1` undoes the last turn); `0` goes back to before the first turn; `k > 0` keeps turns `1..k` of the active path.
`to`	An entry id from `gptr_checkpoints(s, all = TRUE)`; overrides `turn`. Any entry works, including the end of an abandoned branch, which is how redo works.
`restore`	`"all"` (objects, files, session state and the conversation), `"conversation"` (only the conversation) or `"workspace"` (only objects, files and session state; the conversation stays where it is).
`force`	Also restore items that changed after the checkpoint (3-way conflicts).
`preview`	Return the plan instead: a data frame with one row per item (`record`, `turn`, `action`, `checkpointer`, `item`, `restore`, `reason`); nothing changes.
`all`	Also list the turns of abandoned branches.

Details

Restores are 3-way: an object or file that changed after the turn (by you or another session) is kept and reported, unless `force = TRUE`. Reference objects (environments, R6, external pointers), objects over the undo budget, environment variables, loaded namespaces, the random-number state

and effects outside R (network, databases, processes) are not restored. `s$last_rewind$report` lists each object, file and state item once, by its outcome after the whole rewind; items that were not restored raise the warning `gptr_warning_rewind_partial`. Protected files and files that configure R or gptr (`.env`, `.Renviron`, `.Rprofile`, `renv.lock`, keys under `.ssh/`; see gptr_security) are never copied to the checkpoint store, so a turn's edit or deletion of one is not undone. Directories are not tracked: an empty directory a turn created stays. Files outside the project and `tempdir()` are restored only after you confirm each path. After a full restore the model is told nothing (the next request's prefix is byte-identical to the earlier request at the target); after a partial one the next prompt carries a short `<rewind>` block listing what was not restored. Model code may rewind another session only when you approve that call.

Value

`gptr_rewind()` returns `s`, invisibly (the plan when `preview = TRUE`); `s$last_rewind` holds the report and `s$editor_text` the undone prompt, whose preloaded skills are `s$last_rewind$skills`. `gptr_checkpoints()` returns a `gptr_checkpoints` data frame (it prints at most 20 rows).

Options

Options of checkpoints and rewind (`?gptr_options` collects every option):

- `gptr.checkpoint` (`"on"`): `"files"` checkpoints files only, `"off"` nothing (also the settings key `checkpoint`).
- `gptr.undo_capture_max` (`1e8`): bytes; objects up to this size are captured by reference.
- `gptr.undo_max_bytes` (`1e9`): bytes of pre-images held in memory per session.
- `gptr.undo_spill_max` (`2e9`): bytes; the largest object image written to disk.
- `gptr.undo_turns` (`20`): object images older than this many turns are dropped.
- `gptr.checkpoint_disk_bytes` (`2e9`): bytes of blobs and spilled images before a notice.
- `gptr.checkpoint_days` (`30`): unreferenced blobs older than this many days are deleted.
- `gptr.checkpoint_turns` (`100`): file checkpoints older than this many turns are not restored.
- `gptr.checkpoint_track_file_max` (`1e6`), `gptr.checkpoint_track_total` (`1e8`): bytes per file and in all of the baseline taken at the first mutating call.
- `gptr.checkpoint_capture_max` (`5e7`): bytes; a larger changed file gets no stored copy.
- `gptr.checkpoint_scan_budget` (`0.25`): seconds; slower walks switch to one scan per turn.
- `gptr.checkpoint_rng` (`TRUE`): report changes of the random-number state.
- `gptr.checkpoint_close_devices` (`FALSE`): close graphics devices an undone turn opened.

Examples

```
d = tempfile("project")
dir.create(d)
op = options(gptr.project_root = d)
fake = gptr_fake_provider(list(
  list(tool = "r", input = list(code = "x = 1")), "Made x.",
  list(tool = "r", input = list(code = "x = x + 1")), "Added one."))
work = new.env()
```

```
session = peter("Make x", model = fake, mode = auto, envir = work)
session |> peter("Add one to x")
work$x
session |> gptr_rewind()
work$x
session$editor_text
gptr_checkpoints(session, all = TRUE)
options(op)
```

---

gptr_risk *Classify the risk of R code without running it*

Description

gptr_risk() is the advisory static classifier behind gptr's permission prompts. It reads code without evaluating it and labels each effect it can see with a level from 0 (known read-only) to 4 (critical) and a category (read, object_write, file_write, file_delete, network, process, install, dynamic, session, secret, interactive, critical, control, plus unlisted for functions the risk tables do not list). Level 0 is only what the tables know to be read-only; code whose function is computed at run time is level 3. It cannot see S4 methods, compiled code, package load hooks or the files source() reads: it decides when gptr asks you, it is not a security boundary.

Usage

```
gptr_risk(code, envir = NULL, root = NULL)
```

Arguments

code	R code: a character vector, a call or an expression.
envir	NULL or the environment the code would run in. When given, assignments to existing bindings (of envir itself for a plain assignment evaluated in envir; for <<-, assign(), replacement calls and an assignment inside local(), with() or within() also of its enclosing environments up to the global environment) are reported as overwrites with their size (the workspace's size rule in a leaf: each part once, without code or shared environments; promises and active bindings are never forced) and user-defined functions called by the code are classified through their bodies.
root	The project root used for path classes; NULL means the current project root.

Details

Paths are classified relative to root (workspace, temp, outside, protected, control, instructions, critical, url, wildcard, unknown). Calls of peter$sh(), system(), system2() (its env as the start of the command line), processx::run() (also as callr::run()) and sys::exec_wait(), sys::exec_internal() and sys::exec_background() with literal commands are classified with the command table (inst/extdata/risk-commands.csv), peter$sql() by its statements and peter$py() by a token scan. A command the code computes, or a shell.exec() file it computes, is a dynamic row at level 3. A path, command or variable name built with c(), list(),

paste0(), file.path() or another helper of base R counts as one the code computes when the code binds that helper's bare name with =, <- or <<-, as a function's argument or as a for variable, or when the helper found from envir (the global environment when NULL) is not base R's; a helper the code binds otherwise (with assign(), delayedAssign(), makeActiveBinding(), list2env() or attach(), in a list given to with() or within(), or into an environment, e$paste0 = f or .GlobalEnv[["paste0"]] = f) is read as base R's. Plugins and users extend both tables with risk_rule records.

Value

A gptr_risk object: a list with level (integer 0-4), label, categories, flagged (a data frame with columns call, fn, level, category, path, path_class), paths, secret, secret_guard, assigned, dynamic, and the additive fields sizes (bytes of overwritten objects, by name), secrets (guarded secret names), kind, parse_error, calls (every function the code calls, including through the functions it reads in envir) and targets (files: the files the code writes, deletes or downloads to, as permission rules see paths).

Examples

```
gptr_risk("unlink('data', recursive = TRUE)")$level
gptr_risk("summary(mtcars)")
e = new.env()
e$df = data.frame(a = 1:3)
gptr_risk("df = head(df, 2)", envir = e)$flagged
```

---

gptr_router *Define a model router*

Description

A router is a virtual model: route(request, ctx) runs before every request of a session whose model is the router's name and returns a model reference, or list(model, thinking = NULL, state = NULL). A router that errors or exceeds timeout falls back to the default model.

Usage

```
gptr_router(name, route, description = NULL, timeout = 2)
```

Arguments

name	Router name, usable as model = <name>.
route	function(request, ctx).
description	One line for listings.
timeout	Seconds allowed per call.

Value

A spec of class c("gptr_router", "gptr_spec").

Examples

```
gptr_router("cheapest", route = function(request, ctx) "anthropic/claude-haiku-4-5")
```

---

gptr_scrub *Find, and optionally remove, registered secrets in persisted files*

Description

Secrets are redacted when they enter a transcript, a cache or a document, but a value that reached a file before it was registered (for example a key pasted into a prompt and only later loaded with `gptr_env()`) stays there. `gptr_scrub()` scans persisted files for the values of every registered secret and their derived forms and reports where they occur. With `dry_run = FALSE` it rewrites those files, replacing each occurrence with a `[secret:NAME]` marker; this is the only sanctioned rewrite of append-only session files, each of which then gets a `gptr.scrub` entry. A key that reached a model or a commit must still be rotated.

Usage

```
gptr_scrub(paths = NULL, dry_run = TRUE, error = FALSE)
```

Arguments

`paths`	`NULL` for the workspace's sessions, caches, spill files, plans, transcripts and safe relative document paths bound in this project. An explicit character vector of files and directories overrides that scope; directory traversal stays within each root. Binary files and files with existing writer locks (including stale locks) are reported but not rewritten.
`dry_run`	If `TRUE` (the default), only report; if `FALSE`, rewrite the files.
`error`	If `TRUE`, signal an error of class `gptr_error_secret_found` when any file contains a registered secret (for pre-commit hooks and CI).

Value

A data frame with columns `file`, `secret` (the variable name) and `count`; never values. Returned visibly for a dry run, invisibly after a rewrite.

Examples

```
d = tempfile("proj")
dir.create(d)
writeLines("nothing secret here", file.path(d, "notes.txt"))
gptr_scrub(d)
```

gptr_security *Security considerations*

Description

gptr runs language model agents in your current R session. An agent reads files, writes files, evaluates R code written by the model in the environment you give it and starts processes on your behalf. This page describes what gptr does to keep those actions under your control and where its protections end. gptr_egress describes what is sent to model providers.

Everything happens because you asked

When the package is loaded, it only stops the child processes that an earlier gptr session left running after it was killed or crashed, and removes their records in `tools::R_user_dir("gptr", "cache")`. A model is contacted only when you call `peter()` (or a function that you give a prompt), and model-written code is evaluated only in the environment you pass as `envir` (by default the frame that called `peter()`); gptr never assigns into the global environment by itself. Files are written only into paths you pass to a gptr function (for example `to` in `gptr_export()` or `paths` in `gptr_scrub()` with `dry_run = FALSE`), inside the project's `.gptr/` directory once it exists (see `gptr_init()`), inside `tools::R_user_dir("gptr")`, inside the session temporary directory, into files you bound with `gptr_doc()` or confirmed, and by tool calls that the permission mode allowed.

Permission modes

Every tool call passes a permission gate. The default mode is `manual`: actions that the static classifier labels read-only run without asking; every other action it recognizes, including a change to a file or an object, asks first with a one-line prompt. The classifier is advisory, not a security boundary (see below). `edits` also allows file edits inside the project and still asks before R code that is not known to be read-only; `auto` asks only for level-4 actions (for example deleting the project directory or a file outside the project, where a system temporary directory such as `/tmp` counts as outside, moving R's working directory out of the project, or changing gptr's own configuration) and for reads of registered secrets; `yolo`, which only you can select (a project setting or model code in another mode cannot), also runs gptr's own configuration functions, writes to R startup files such as `~/.Rprofile` (which can change gptr's options in the next R process) and secret reads without asking, and still asks before deleting protected files, files outside the project or the project itself, moving the working directory out of the project, `rm(list = ls())`, `options(gptr.*)` and sending a secret over the network (see Secrets below); `plan` changes nothing, though a call it reads as read-only can still use the network or start a process: `gptr_models(refresh = TRUE)` downloads the model catalog, and `gptr_providers()` with `check = TRUE` or `check_login = TRUE` contacts the providers or starts the `claude` and `codex` status commands. In every mode `r` code that calls `q()`, `readline()` or another function that would end, pause or hang your R session is refused without a question. In `auto` and `yolo` mode a `system()`, `system2()`, `pipe()`, `processx::run()` or `data.table::fread(cmd =)` call (on Windows also `shell()` or `shell.exec()`) whose command the code computes is refused (see below). When nobody can answer a question (Rscript, knitr, a scheduled job) the run stops with status `blocked` and a `gptr_error_permission` condition that says how to allow the action, unless you set `options(gptr.noninteractive_ask = "deny")`. Model code cannot switch the gate off or loosen it during a run: the safety options are copied when the run starts. Only the option

gptr.unsafe_no_permissions, set by you outside a run, removes the gate, and it is meant for disposable sandboxes.

Changing gptr's own configuration includes model code that sets or unsets an environment variable gptr reads as configuration: a key or base-URL variable of a provider gptr has registered (such as AWS_REGION, which places Bedrock's endpoint), a variable gptr keeps for the claude or codex command-line tool (its login, its folder or the path of Git Bash, such as CODEX_HOME), or one that moves its configuration folder or your home directory, such as HOME; gptr_options lists them. The classifier reads such a change in Sys.setenv(), Sys.unsetenv(), withr::local_envvar() and withr::with_envvar(), where a name it cannot read counts as one (also one built with paste0() or another helper of base R whose bare name the code binds with =, <- or <<-, as a function's argument or as a for variable, or that your session holds as other than base R's, and the names passed to a setter handed to lapply(), Map(), do.call() or another function the classifier knows calls it, such as sapply() or purrr::map(); do.call() of a list() or c() written out and rlang::exec() are read as the call they make, so do.call(Sys.setenv, list(TZ = "UTC")) is Sys.setenv(TZ = "UTC")), in readRenviron() and dotenv::load_dot_env(), and in an os.environ write in the code of peter$py() or peter$knit("python", code) that names one in a string or an uppercase word, or whose name (a subscript, or the first argument of putenv(), unsetenv() or pop()) is not a plain string literal. It does not read code in a file R runs (source(), sys.source(), eval(parse(file =))); Sys.setenv(), Sys.unsetenv(), withr::local_envvar() or withr::with_envvar() reached by a computed name, through a list or an environment, or handed to a function the classifier does not know calls it (get("Sys.unsetenv"), baseenv()$Sys.setenv, asNamespace("withr")$local_envvar, with(list(f = withr::with_envvar), ...), assign("f", Sys.unsetenv), identity(withr::local_envvar)); a name built with a helper the code binds otherwise (with assign(), delayedAssign(), makeActiveBinding(), list2env() or attach(), in a list given to with() or within(), or into an environment, e$paste0 = f or .GlobalEnv[["paste0"]] = f), which is read as base R's helper builds it; a Python write that reaches os.environ by another name (e = os.environ, getattr(os, "environ")) or through ctypes, or that passes os.environ.update() a mapping the code computes, a mapping whose names are not plain string literals ({'\x48OME': d}, {'H' 'OME': d}) or keyword names not all in upper case (on Windows os.environ.update(home = d) sets HOME); or Python run through 'reticulate': such code is level 3 or lower, which auto runs without asking.

What the gate cannot do

The risk levels come from a static classifier (gptr_risk()). It is advisory and it is not a security boundary: R code can compute function names, call compiled code or load packages whose effects the classifier cannot see, and R has no sandbox for code that runs inside your session. The classifier reads code as written and does not run it, so code that builds a call, a name, a path or a command at run time, or reaches a function by another name, is read only as far as the classifier can see. Each form this page names as not read is of that kind, or is code in a file or a package, code another interpreter runs, or a call or argument the classifier has no rule for. A function it has no row for, such as gptr_step() or gptr_wait(), reads level 1, which manual asks about and auto runs (gptr_export() called from model code is refused when it runs). The code in the braces of a 'glue' or 'cli' template (glue::glue("{unlink(d)}"), stringr::str_glue(), cli::cli_text()) runs when the text is formatted and is not read. Deny and ask sh: rules see a command computed at run time only in a peter$sh() or peter$bg() call, which checks the command when it runs. So in auto and yolo mode a system(), system2(), shell(), shell.exec(), pipe(), processx::run() or data.table::fread(cmd =) call whose command (for system2() also its arguments and env) is not a string the classifier can read, such as cmd = paste("rm", "-rf", "build"); system(cmd),

is refused, and the model is told to run the command with `peter$sh()`; `manual` and `edits` ask. An allow rule such as `r(fn:system)` lets such calls run, unchecked by `sh:` rules. Neither the rules nor the refusal read what a shell reads when it runs: its standard input (`system("sh", input = cmd)`, `peter$sh("sh", input = cmd)`), variables it expands (`sh -c "$X"`, `eval $X`) or a script file. The refusal covers only calls the classifier resolves to these functions: a function reached by a computed name or through a list or an environment (`get("system")(cmd)`, `baseenv()$system(cmd)`) or handed to a function of your own (`g(system)`), `eval(parse())`, a sourced file, package code and other command runners such as `processx::process$new()` or `sys::exec_wait()` still run a computed command unchecked. A literal command is read when it is given to `system()`, `system2()`, `peter$sh()`, `processx::run()` (also as `callr::run()`), `sys::exec_wait()`, `sys::exec_internal()` or `sys::exec_background()`, not when it is given to `processx::process$new()`, which is level 3. A command line in a quoted string is read as one the code runs, so `system("echo 'rm -rf /'")` reads level 4, as `system("ssh host 'rm -rf ~'")` does.

Paths are placed in the project: a relative path in R code is read in the project directory, and a command in the directory it runs in (after a `cd` or `env -C`, or the `wd` of `peter$sh()` or `processx::run()`), the code a shell or Python runs there (`sh -c`, `cmd /c`, `pwsh -Command`, `python3 -c`) included; running a command in a directory outside the project reads that directory, so `manual` asks. `setwd()`, `withr::local_dir()`, `withr::with_dir()` and `xfun::in_dir()` to a directory outside the project and R's session temporary directory (which `tempdir()`, `tempfile()`, `withr::local_tempdir()` and `withr::local_tempfile()` name), and `setwd()` and `withr::local_dir()` to one the code computes (such as `here::here(d)`; `here::here()` of literal parts is read in the project), are level 4, because a relative path would then name a file outside the project, and so are `file.symlink()`, `file.link()`, `Sys.junction()` and `fs::link_create()` of a path outside the project, as `ln` of one is; `chdir()` in the code of `peter$py()` or `peter$knit("python", code)` (`os.chdir()`, `os.fchdir()`, `contextlib.chdir()`), which moves R's working directory too, is read as `setwd()` is. After the code has moved R, a relative directory given to these functions is read in the directory R is in, so `setwd("data"); ...; setwd("..")` returns to the project, as do `os.chdir("data")` and `os.chdir("..")` in Python when each is a statement of its own on a line with no indentation, no `:` and no bracket left open, in code with no triple-quoted string (after any other `chdir()` a relative directory is read both in the directory R was in and in the one that `chdir()` names); before any move it is read in the project. Python comments are not read as code, except in code with a triple-quoted string, which is read with its comments. Python's restore idiom `cwd = os.getcwd(); os.chdir("data"); os.chdir(cwd)` returns to a directory the code computes, so it asks even in `auto` and `yolo`. `getwd()` and the directory `setwd()` returns are read as the one the code has moved R to, starting from R's working directory when the classifier reads the code. A function body and the code of `local()` are read as code of their own: the names they assign are followed (a name assigned with `<<-` or `assign()` counts, outside them, as one the code computes), a function's other names count as ones the code computes, and R starts where the code started or has moved it to. So `owd = setwd("data"); ...; setwd(owd)` stays in the project at the top level of the code, in a function (with `on.exit(setwd(owd))`), in `local()` and in `tryCatch(finally = setwd(owd))`. A directory the code computes that R moves back from is level 3: that of `withr::with_dir()` or `xfun::in_dir()`, and one saved with a plain `owd = setwd()` or `owd = getwd()` in a branch or a loop (`if (p) owd = setwd("data")`), but only for `setwd(owd)` in the scope that saved it and only when the code binds that name no other way anywhere (to another value, as a loop variable or a function's argument, with `<<-`, `assign()` or `list2env()`, or by an assignment in `eval()` or `with()`); otherwise `setwd(owd)` to a directory the code computes stays level 4. A relative directory after `withr::with_dir()` or `xfun::in_dir()` counts as one the code computes. A drive-relative path such as `D:data` counts as outside on every OS. The classifier does not follow the working directory changed in other ways (such as

usethis::local_project(), get("setwd"), the wd of callr::r(), or env given -C among other short options, env -iC DIR, or as an abbreviated --chdir, env --chd=DIR), nor read the code another interpreter runs (Rscript -e, perl -e, node -e, pwsh -CommandWithArgs or -cwa), a path the shell computes (rm "$f" in a loop, xargs rm) or code handed to sh -c or bash -c as one word whose spaces are escaped with a backslash or written $IFS (sh -c rm\ -f\ file): such a command is level 3, which auto and yolo run without a question whatever it deletes. The wd of processx::run() is read when the call, or a name the code assigns, holds it as a string; a wd the code computes is read as the project directory. A delete function handed to lapply(), Map() or purrr::walk() reads its paths as ones the code computes, level 3, so auto runs lapply("~/notes", unlink) without a question.

In auto and yolo mode a model can delete or overwrite files and objects. Use them only in projects you can restore (version control, backups, gptr_rewind() for recent turns), and run untrusted prompts or untrusted projects in a disposable container or virtual machine.

Sub-agents, workers and child processes

Inline sub-agents run in your session and see its objects. The worker backend runs a sub-agent in a separate R process that starts from gptr's reduced child environment (it does not read your ~/.Renviron), but that process runs under your user account with your file permissions: the worker backend is not an isolation boundary. Its permission requests are sent to your session and classified again there. Your session reads a worker's result file with readRDS(): on R before 4.4 a crafted result file can run code in your session, and the objects a worker returns are R objects by design. gptr_coach() runs every run and every coach as an unattended session in its own R process under your account, which is not a sandbox either; a fence keeps a coaching run's writes in its own folder as far as the classifier sees them, and Codex's own shell is off in coaching sessions. Elsewhere, on the codex route Codex's own shell runs commands in every mode without passing gptr's gate, rules or secret guard: in plan, manual and edits its sandbox is read-only, so it can still read any file your account can read, and in auto and yolo it also changes and deletes files inside the project within its sandbox. MCP servers, the claude and codex command-line tools, shell commands started with peter$sh() and Shiny artifacts are ordinary processes of your account as well; artifacts start without any registered secret and listen on the loopback interface, and each launch gets an access token in its address (on Windows only when 'openssl' is installed; otherwise gptr says that the artifact has no token).

Untrusted content

Model replies, tool results, file contents, MCP results and project instruction files (AGENTS.md, CLAUDE.md, .gptr/vignette.Rmd) are treated as data. Text from these sources is never used as a formatting template, so braces in a reply cannot run R code. A project you cloned is untrusted until you trust it with gptr_trust(trust = TRUE) (see gptr_trust()), or answer yes when gptr_init() asks: until then its settings can only tighten permissions, its extensions, plugins and MCP servers do not run, and its instruction files are shown to the model as information rather than commands. When trusted files change after a pull, gptr asks again.

Secrets

Keys are held in a private vault and appear in sessions, logs, errors and permission questions only as markers such as [secret:ANTHROPIC_API_KEY]. Values are redacted before anything is written or sent: the session files, documents, caches, spill files, wire logs and the output of child processes.

Their URL-encoded, JSON-escaped and base64 forms are redacted too; a copy the code changes in another way, such as reversed or cut, is no longer recognized and is not redacted. The console shows what model code prints with registered secrets redacted where it indents that output under the call, and so are the warnings it shows; where it does not indent it (a console that cannot redraw a line, or `gptr.verbose` below 2), R prints the output itself, a registered secret included, on your screen only. gptr never reads the credential files of other tools. Reading a registered secret from model code asks even in `auto` mode: `Sys.getenv()` of its name in any letter case, or of a variable the code only ever sets to strings one of which is its name; `$NAME` or `${NAME}` in a command line (`system()`, `processx::run()`, `peter$sh()` and the others above) and `os.environ` or `getenv()` in Python; an environment dump (`Sys.getenv()`, `ps::ps_environ()`); and a read of a `.env` or `.Renviron` file (`.env`, `.env.local` or `name.env`), of a file named `auth.json` (gptr's credential store) or of a file `gptr_env()` loaded, by a function that reads its text such as `readLines()`, or in a command line or Python. So does handing one to a child process through the `env` of `peter$sh()` or `peter$script()`, by its name or in a value, unless the same code reads that secret by name: the question for that read, or an `r(secret:NAME)` rule allowing it, covers the hand-over too. A name the code computes when it runs (with `paste0()`, through `get()` or `do.call()`, as a function's argument, or in a variable it sets otherwise or only to strings naming no secret) is read at level 3, which `auto` runs without asking. Such a file read through `lapply()` or another function that calls the reader (`lapply(".env", readLines)`), or read from a copy (`file.copy(".env", "k.txt")`, then `readLines("k.txt")`), is not read as a secret read either. Sending a secret over the network asks in every mode, `yolo` included: code that reads a secret, here or in an earlier call, or hands one to a child through `env`, and calls an R network function such as `download.file()` or `httr2::req_perform()`, or holds a string naming a network program such as `curl`, `wget` or `ssh` and starts a process with `system()`, `system2()`, `processx::run()`, `peter$sh()`, `peter$bg()` or `peter$script()`; a program name the code builds from pieces is not read. A key registered after it already appeared in a file (for example pasted into a prompt) can be removed from the files with `gptr_scrub()`; run `gptr_scrub(error = TRUE)` before committing to fail when any registered secret remains.

Data at rest and protected health information (PHI)

gptr keeps what it needs to resume and replay work. Review what you commit when the data are personal or clinical:

- `.gptr/sessions/` (full conversations, including printed results) is excluded from git by the `.gptr/.gitignore` that `gptr_init()` writes.
- `.gptr/cache/s1/` holds System 1 answers and is committed by default so that a script replays without model calls. It stores a salted hash of each input and a hash of each question, never their text, but the answers themselves (a label and its probability) remain, and the salt is committed with the cache, so anyone with the repository can test guesses of short inputs. For PHI, stop committing it with `gptr_config(cache_commit = list(s1 = FALSE, s2 = FALSE), .scope = "project")` (gptr then keeps a `.gitignore` in `cache/s1/`), or clear it with `gptr_cache("clear", "s1")` before committing.
- `.gptr/cache/s2/` (answer text for replay), `.gptr/transcripts/` (console transcripts), `.gptr/checkpoints/` (pre-images of objects and files) and the data snapshots of artifacts (`.gptr/artifacts/<id>/v<n>/data/`) are excluded from git by default. The checkpoints never hold a copy of a protected file or one that configures R or gptr (`.env`, `.Renviron`, keys under `.ssh/`).
- `.gptr/plans/` (plans proposed in `plan` mode) and the `app.R` of each artifact are committed by default; they can quote data values the model saw.

- Documents you record into keep the model's code and up to `gptr.doc_output_lines` lines of printed output per step, which can contain data values. A console turn's block also keeps, as `#<` lines, what its prompt was sent with, whatever that option says: the head of each `@file` (up to 40 lines) and the `!` code notes with their printed output. Review them before sharing.
- The block of a recorded `gptr_coach()` call keeps the coaching it chose (its note, skills, review and rules), which coaches write from your runs and which can quote data values they saw.

Local servers

`gptr_mcp_serve()` exposes the live session to other agents over HTTP on `127.0.0.1` only, with a random bearer token and an `Origin` check, and every call passes the permission gate. Any local process that learns the token can call the served tools while the server runs; stop it with `gptr_mcp_serve(stop = TRUE)` when you are done.

See Also

gptr_egress, gptr_options, `gptr_permissions()`, `gptr_trust()`, `gptr_scrub()`, `gptr_risk()`.

---

gptr_sessions *Stored and live sessions*

Description

`gptr_sessions()` lists the sessions of the workspace store (`.gptr/sessions/`, or `tempdir()/gptr/sessions/` without a workspace), newest first. It reads only the header and the last lines of each file.

`gptr_resume()` returns the live object when one exists in this process; otherwise it rebuilds the session from its JSONL file (leaf = the last entry; the mode it last ran with, or the `mode` setting for a file that records none; status `idle` when the tail is a final answer or, for a team or fan-out, an idle settlement, else `interrupted`) with home `envir`. A rebuilt fork always gets a fresh overlay of `envir`. A detached copy (from `saveRDS()`, a knitr cache or callr) is attached under the split-brain rules. `block =` returns the session that a document replay bound to that block. Resuming restores conversation state, not a complete analysis workspace, and does not rerun the R tool code. Make required objects available in `envir`; use document replay when the intention is to execute the recorded analysis again.

`gptr_last()` returns the most recently active session of this R process. It is held strongly (it survives `gc()`), so a session whose call was interrupted before its result was assigned is not lost.

Usage

```
gptr_sessions(project = TRUE)

gptr_resume(x = NULL, envir = parent.frame(), block = NULL, child = NULL)

gptr_last()
```

Arguments

project	TRUE: the stored sessions; FALSE: also the live sessions of this process that are not in that store.
x	NULL (the most recently updated stored session), a session id, a file path, or a detached gptr_session.
envir	The home of a rebuilt session.
block, child	A document block id (and a team member name): the session replay bound to that block in this process, or gptr_error_replay_unbound; never a fallback to envir.

Value

gptr_sessions() returns a gptr_sessions data frame: id, file, created, updated, turns, model, status, title (the first prompt, 60 characters), live. gptr_resume() returns a gptr_session: the live object when one exists in this process, otherwise one rebuilt from its session file. gptr_last() returns the most recently active session of this process, or NULL when no session was created in this process.

Examples

```
session = peter("hi", model = gptr_fake_provider(list("hello")))
identical(gptr_last(), session)
gptr_sessions()
identical(gptr_resume(session$id), session)
```

---

gptr_skills *Discovered skills and agents*

Description

gptr_skills() lists the Agent Skills gptr can see: the project's .gptr/skills, .agents/skills and .claude/skills, the user's skill directories, the skills of gptr and of attached packages, and those of enabled plugins. Project skills are listed even when the project is not trusted (their source reads project (untrusted)), but only trusted skills enter the catalog the model sees. gptr_agents() lists agent definition files the same way. Both only read files.

Usage

```
gptr_skills(scope = c("all", "project", "user", "packages"))

gptr_agents(scope = c("all", "project", "user", "packages"))
```

Arguments

scope	One of "all", "project", "user" or "packages".

Details

A skill is a directory containing `SKILL.md` with a YAML `description` and optional `name`. Its body is read on demand, or preloaded with `peter(skills = "name")`. `disable-model-invocation: true` hides it from automatic catalog selection, while explicit preloading remains available. `allowed-tools` is metadata, not a permission rule. An extension or a harness can also register a skill with `gptr_spec("skill", name, description = "...", text = "...")`; such a skill has no files, so `gptr_skills()`, which reads skill files, does not list it.

Value

`gptr_skills()`: a `gptr_skills` data frame with columns `name`, `description`, `source`, `path`, `tokens` (catalog cost) and `visible`. `gptr_agents()`: a `gptr_agents` data frame with columns `name`, `description`, `model`, `backend`, `source` and `path`.

See Also

gptr_agent, gptr_plugins, gptr_trust; `vignette("skills-and-plugins", package = "gptr")` for skills, templates and agent files.

Examples

```
gptr_skills("packages")
gptr_agents()
```

`gptr_source` *Source a history document, regenerating stale blocks without running the old code*

Description

Evaluates `file` top-level expression by expression in `envir`, like `source()`. An expression that names `peter()` or `gptr_coach()`, or calls a harness bound in `envir`, keeps its source references, which locate the call; the others run without them, so the functions and calls they make hold no part of the document, as at the console. A `peter()` call whose block is fresh replays without calling a model and the block's code then runs as ordinary R. A call whose block is stale (its prompt or the values interpolated into it changed), or every call under `replay = "live"` (an undone block runs without being rewritten; `replay = "record"` regenerates it), asks the model again and rewrites its block in place, and the old block is skipped, which base `source()` cannot do. Only that block is regenerated: a later turn of the same session keeps its block, answered in the old turn's context, so re-ask it with `replay = "live"` on that call when it depends on the turn that changed. Only blank lines may separate a call from its blocks (see `gptr_doc()`). A console turn is asked again with what the console sent with it, from its block's `#<` lines (the `@file` heads and `!` notes; see `gptr_doc()`): the request is the same even when the file changed or is gone. It is asked with the model and in the mode its statement names, else those its session has from the turns before, else the `model` and `mode` settings of the session that re-asks: a statement names them only where they differed from what it gets without them (see `gptr_doc()`). A named model must be available there; `model` asks a `peter()` or harness call with another one. A re-ask does not send the `## Steer:` and

`## Follow-up:` lines of a turn or the text of an approval answered `n <text>`, and the new block does not keep them. Direct R lines a transcript turned off are comments and do not run. A block is written only with write consent (see `gptr_doc()`); a call that cannot be recorded still runs, and under `replay = "record"` a notice says that gptr did not record it.

Usage

```
gptr_source(
  file,
  replay = getOption("gptr.replay", "auto"),
  envir = parent.frame(),
  echo = FALSE,
  model = NULL
)
```

Arguments

`file`	Path of an `.R` document.
`replay`	Replay mode for the calls in the file: `"auto"`, `"replay"`, `"live"` or `"record"`. A call's own `replay =` argument wins. Left missing, each call resolves its mode as `peter()` does: the `gptr.replay` option, then the `GPTR_REPLAY` environment variable, then the `replay` setting, then `"auto"`.
`envir`	Environment in which the expressions are evaluated.
`echo`	`TRUE` prints each expression before it is evaluated.
`model`	`NULL`, or a model name or reference string (`"sonnet"`, `"anthropic/claude-sonnet-5-5"`, or one of a provider you registered), not a provider spec object: every `peter()` or harness call the file asks again (each one under `replay = "live"`) uses it instead of the model it names or its session has, so a recorded analysis can be re-asked with your own model. A `gptr_coach()` call and a team or fan-out call (`agents =`, `parallel =`) keep the models they name. A block that replays asks no model.

Details

Replay executes the recorded R code, including its ordinary side effects. It does not freeze package versions, supply external inputs or validate the analysis. This function accepts `.R` scripts; render `.Rmd` and `.qmd` with their report tools, and execute `.ipynb` through its R kernel. Use `gptr_blocks()` to inspect any supported format.

Value

Invisibly, the `gptr_blocks` listing of the file after the run with an extra column `action`: `replayed`, `regenerated`, `ran`, `skipped`, or `NA` for blocks no call touched. A stale block whose call ran but could not be rewritten (for example without write consent) is `ran` and keeps its status.

Examples

```
f = tempfile(fileext = ".R")
writeLines(c("x = 1", "y = x + 1"), f)
gptr_source(f, replay = "replay", envir = new.env())
```

---

gptr_spec *Build a spec of any registered kind*

Description

The general constructor behind the eleven spec constructors: it builds a spec of `kind` from its fields and validates it with the kind's validator. Use it for kinds without a constructor (`env_alias`, `setting`, `service`, `preset`, ...) and for kinds defined by plugins. Unknown fields are kept (forward compatibility); a wrong type of a known field is an error naming the field.

Usage

```
gptr_spec(kind, name, ...)
```

Arguments

`kind`	The kind name, one of the `kind.<name>` features of `gptr_api()`.
`name`	The record name within the kind.
`...`	The kind's fields, named.

Value

A spec: a list of class `c("gptr_<kind>", "gptr_spec")`.

Examples

```
gptr_spec("env_alias", "SLACK_BOT_TOKEN", aliases = "slack-token")
gptr_spec("setting", "panel.size", default = 3L, description = "Reviewers per panel")
```

---

gptr_steer *Steer a session*

Description

The one enqueue function behind the pipe into a running session, the pause menu and `ctx$send()`. A steer reaches the model after the current tool results, as "The user sent this message while you were working: ..."; a follow-up when the agent would otherwise stop. On an idle session the item is taken when the next run starts. The queue lives in the R process; items still queued when it ends are not saved. In a script replayed from its document, a steer the replayed block already recorded is not sent again.

Usage

```
gptr_steer(s, text, as = c("steer", "follow_up"))
```

Arguments

s	A session.
text	The message (secrets are redacted before it is queued).
as	"steer" or "follow_up".

Value

s, invisibly, at once.

Examples

```
session = peter("Summarize mtcars", model = gptr_fake_provider(list("ok")),
  .run = FALSE)
gptr_steer(session, "Use only the mpg column", as = "follow_up")
```

---

gptr_step *Advance a session*

Description

Starts the queued input of an idle session (built with .run = FALSE, or given a follow-up with gptr_steer()) and pumps it until turns model turns have ended or the run settles. A running session is advanced the same way; a waiting background session is left as it is (gptr_wait() continues it, see gptr-background).

Usage

```
gptr_step(s, turns = 1L)
```

Arguments

s	A session.
turns	The number of turns to advance (Inf for all).

Details

.run = FALSE builds a session without making its model request; gptr_step() or gptr_wait() starts it. Supply an explicit workspace with envir = when results must remain available outside the function that creates the session.

A session left running after its turns is not paused: its run continues, tool calls included, during any later blocking gptr call at top level, such as another peter() call, gptr_source(), or gptr_wait() or gptr_step() on another session. In mode "auto" such a call can thus run the session's R code and write its files. Call gptr_wait() on the session to finish its run first, or gptr_cancel() to stop it.

Value

s, invisibly. A run that settles in status error, blocked, budget or max_turns signals the condition documented in peter().

Examples

```
session = peter("Plan the analysis", model = gptr_fake_provider(list("Plan: ...")),
  .run = FALSE)
gptr_step(session)
session$turns
```

---

gptr_tailor *Build a harness: peter with your own fixed arguments and coaching*

Description

A harness is a function you call exactly like peter(). gptr_tailor() makes one from peter or from another harness, without calling any model: it fixes peter() arguments such as mode, model, tools or extensions, it can carry your own hard rules, and it can carry *coaching*, plain data that gptr itself turns into a note added to peter's instructions, skills peter can open, a review that asks peter to check its work once or twice before it stops, and hard rules. Each call of the harness is the same call as peter() with those arguments added, so objects, envir, {name} interpolation, gptr_last(), pipes, lapply() and do.call() behave as with peter().

Usage

```
gptr_tailor(x = peter, ..., rules = NULL, coaching = NULL, dir = NULL)
```

Arguments

x	peter or a harness to start from.
...	Named peter() arguments to fix, resolved now as peter() resolves them (so mode = auto is "auto"); prompt is each call's task and cannot be fixed. plugins cannot hold "+name" or "-name" filters, which would change the whole R session.
rules	NULL to keep your rules of x, else your own hard rules, which no coaching changes: list(never, require). never holds up to 8 list(rule, why): a call that matches rule is refused in every mode, also in sub-agents, with "denied by your rule". require holds up to 4 list(id, any, contains, text) or list(id, made, text): when peter is about to finish without a matching call in that run, text is sent to it, at most twice, and the run ends with a notice naming id. A rule or an any pattern is a permission rule (see gptr_permissions()) of the kind r(fn:name), r(category:name), r(sh:glob), r(sql:keyword) or a path glob such as write(data/raw/**); it matches when any call in the code does, and fn: names every function the code calls. A require pattern names only functions or categories rated below level 3. contains is text the call's code must hold; made is "plot" or "file:<glob>" (a file inside the project changed during the run). list() drops your rules.

coaching — NULL to keep the coaching of x, else list(note, skills, review, rules): note one string of at most 8,000 characters; skills a list of list(name, description, text) with names such as "survey-weights", descriptions of at most 1,024 characters and 64 KiB in all; review list(text, rounds) with at most 2,000 characters and 0 to 2 rounds (default 1); rules as rules above, refused with "denied by the coaching rule". The texts may not hold control characters, zero-width or other format characters (so text in a script or an emoji sequence that needs a zero-width joiner cannot appear), tags of gptr's own prompt blocks, or gptr's wording for a message from you or an extension.

dir — A folder to write the harness to for people to read or keep in git (HARNESS.md, NOTE.md, REVIEW.md, RULES.md and skills/<name>/SKILL.md). gptr never reads it back.

Details

Arguments you give in a call win over the fixed ones, except that tools, extensions, plugins and skills add to the fixed values and .opts entries replace them by name. The same rule applies when you tailor a harness again; NULL drops a fixed single value.

To share a harness, save it with saveRDS(). Treat a harness from someone else like an R script from them: functions among its fixed arguments, and the extension files it names, run when you call it. Load it with gptr_tailor(readRDS(file)), which rebuilds it, checks its coaching and shows the code, extensions and plugins it fixes and any system prompt it replaces.

Value

A harness, a function of class gptr_harness with a new id and an empty record (invisibly when dir is given). h$coaching and h$record show its coaching and record, and h$report the last full coaching report of this R session.

See Also

gptr_coach() to improve the coaching by running your task, and peter().

Examples

```
fake = gptr_fake_provider(list("The data has 32 rows.", "Checked: 32 rows."))
count_peter = gptr_tailor(peter, model = fake, coaching = list(
  note = "Count rows with nrow(), never by eye.",
  review = list(text = "Run the count once more before you answer.", rounds = 1)))
count_peter
session = count_peter("How many rows does the data have?", mtcars)
session$text
```

gptr_tool *Define a tool*

Description

A tool is one capability the model can use: declared directly in the request's tool array (`exposure = "direct"`), callable from R code as `peter$<namespace>$<name>()` (`"r"`, one signature line in the prompt), found through `peter$search()` (`"deferred"`) or callable only by gptr code (`"hidden"`). Give `execute` (`function(input, ctx)`, the direct-tool form), `fun` (an R function whose formals match the schema properties, the member form), or both. A direct tool with only `fun` gets a generated `execute` that prints the value within `output_tokens`; an `r` member with only `execute` gets a generated `fun`.

Usage

```
gptr_tool(
  name,
  description,
  parameters = NULL,
  execute = NULL,
  fun = NULL,
  exposure = c("direct", "r", "deferred", "hidden"),
  namespace = NULL,
  execution = c("sequential", "concurrent"),
  risk = NULL,
  snippet = NULL,
  guidelines = NULL,
  signature = NULL,
  output_tokens = NULL,
  record = TRUE,
  available = NULL,
  annotations = list()
)
```

Arguments

`name`	Tool name, `^[a-zA-Z0-9_-]{1,64}$`.
`description`	Model-facing description; direct tools stay under 400 estimated tokens.
`parameters`	A JSON Schema list with `type = "object"`, a `function(ctx)` evaluated once when a session freezes its prompt, or `NULL` to derive the schema from `fun`'s formals.
`execute`	`function(input, ctx)` returning a `gptr_tool_result()`, text, a list or `NULL`.
`fun`	The R-callable form of the tool.
`exposure`	One of `"direct"`, `"r"`, `"deferred"`, `"hidden"`.
`namespace`	Namespace of an `r` member (plugins must set it to their package name).

execution	"sequential" (tools that evaluate R or write files) or "concurrent".
risk	function(input, ctx) returning a risk level, or NULL.
snippet	One line shown in the prompt's tool list (direct tools).
guidelines	Bullet lines added to the prompt's rules (direct tools).
signature	Catalog line for r members; NULL derives it from the schema.
output_tokens	Budget for printed results; NULL uses the gptr defaults.
record	Keep calls of this member in recorded code.
available	function(ctx) deciding at freeze whether a direct tool is offered.
annotations	Named list: read_only, destructive, idempotent, open_world, requires_user (MCP readOnlyHint-style names are mapped).

Value

A spec of class c("gptr_tool", "gptr_spec").

See Also

gptr_register, gptr_check; vignette("extending-gptr", package = "gptr") for complete tool and policy examples.

Examples

```
gptr_tool("nrow_of", "Number of rows of a data frame in the session",
          fun = function(name) nrow(get(name, envir = globalenv())),
          exposure = "r", namespace = "demo")
```

---

gptr_tool_result *Build a tool result*

Description

What a tool's execute() returns: text and images for the model, details kept in the transcript but never sent to a model, and value, the R object handed to R callers of the member (kept in memory only, never persisted).

Usage

```
gptr_tool_result(
  text = NULL,
  images = NULL,
  details = NULL,
  is_error = FALSE,
  value = NULL
)
```

Arguments

text	Character vector, joined with newlines into one text block, or NULL.
images	List of image blocks, PNG file paths or raw PNG vectors, or NULL.
details	Named list of details, or NULL.
is_error	TRUE when the result reports a failure.
value	Any R value returned to R callers.

Value

A gptr_tool_result list with content, details, is_error, value, spill, out_id, truncated and terminate.

Examples

```
gptr_tool_result("3 rows", details = list(n = 3L), value = 3L)
```

---

gptr_trust *Record or read whether a project is trusted*

Description

Trust decides whether gptr applies or runs what a project directory contains: project settings beyond tightening, .gptr/extensions/ and project plugins, project MCP servers (.gptr/mcp.json and the .mcp.json, .codex/config.toml, .cursor/mcp.json, .vscode/mcp.json and .pi/mcp.json of other harnesses), SYSTEM.md and APPEND_SYSTEM.md, project .env discovery, provider base-URL overrides, project skills (.gptr/skills/, .agents/skills/, .claude/skills/ and relative skills.paths entries), prompt templates (.gptr/prompts/), the tool and model fields of project agent files, and the authority of project instructions. The decision is kept in tools::R_user_dir("gptr", "config")/trust.json with a fingerprint of the trust-gated files; when they change, the project counts as untrusted again until you confirm. Called from model code during a run, recording a decision is refused.

Usage

```
gptr_trust(path = ".", trust = NULL)
```

Arguments

path	A directory inside the project; its project root is used.
trust	NULL to read the decision, TRUE or FALSE to record one.

Value

With trust = NULL, the recorded decision: TRUE, FALSE or NA (undecided, also when a trust-gated file changed since the project was trusted). Otherwise the previous recorded decision, invisibly.

Examples

```
d = tempfile("proj")
dir.create(d)
gptr_trust(d)
unlink(d, recursive = TRUE)
```

`gptr_usage` *Token usage and cost*

Description

Aggregates the usage rows of sessions (children included, each request counted once) and, for `x = NULL`, of every live session of this process plus the process System 1 log. Reads only.

Usage

```
gptr_usage(
  x = NULL,
  by = c("session", "agent", "model", "route"),
  detail = FALSE
)
```

Arguments

`x`	A `gptr_session`, a list of sessions, or `NULL` (every live session of this process and the System 1 log).
`by`	Grouping of the summary: `"session"`, `"agent"`, `"model"` or `"route"`.
`detail`	`FALSE`: a `gptr_usage` data frame (`group`, `requests`, `input`, `output`, `cache_read`, `cache_write`, `cost`) with attribute `totals`; `TRUE`: the `gptr_ledger` per request and context component (`request_id`, `component`, `tokens`, `cached`) of the sessions' own requests, session by session in request order (a child's requests are in the child's ledger).

Details

Unknown usage stays unknown: a token count or cost that a provider did not report is `NA`, never zero, so a group or total that includes it is `NA`. The footer prints such tokens as `unknown` and adds the known cost to a count of the requests of unknown cost (`$1.2000 + 1 request of unknown cost`). A known zero, such as the metered charge of a local model, stays zero. Rows without a session, agent, model or route (process-level System 1 calls) form the `NA` group. On `claude-cli` models, billed to a Claude plan, the dollar figures are the CLI's API-equivalent estimate, not a charge; cost limits use them.

The detailed component ledger contains estimates; its sum can differ from provider-reported usage. A run's token budget checks estimated next-input tokens and recorded usage before requests; cost limits check recorded cost. The next output or an unknown final charge can exceed a remaining allowance, so these stopping policies are not billing reservations. Teams and fan-outs count child requests against their shared root budget.

Value

A `gptr_usage` or `gptr_ledger` data frame.

Examples

```
gptr_usage()
session = peter("hi", model = gptr_fake_provider(list("hello")))
gptr_usage(session)
```

`gptr_wait` *Wait for sessions to settle*

Description

Starts idle sessions that have queued input, then pumps until no session is running or waiting, or `timeout` seconds have passed. Every session's checks (egress, replay, its model's provider and key) pass before any is started, so a refusal starts none. On timeout the sessions keep their `running` status and no condition is raised. A background session waiting for approval continues only here, and you are asked as usual (see gptr-background).

Usage

```
gptr_wait(x, timeout = Inf)
```

Arguments

`x`	A session or a list of sessions (a team or fan-out session counts as one).
`timeout`	Seconds to wait.

Details

`timeout` limits reactor polling; it does not interrupt R code. Request preparation and synchronous callbacks finish their current step before the deadline is checked again, so a call can return after the specified time.

A timeout is not cancellation. Inspect `$status` and use `gptr_cancel()` if you want to abort active work. In Rscript and rendered reports, a blocking call such as this is how experimental background sessions progress when there is no idle console.

Value

`x`, invisibly. For a single session, a terminal status signals its condition.

Examples

```
fake = gptr_fake_provider(list("a"))
runs = list(a = peter("one", model = fake, .run = FALSE),
            b = peter("two", model = fake, .run = FALSE))
gptr_wait(runs, timeout = 10)
vapply(runs, function(x) x$status, "")
```

peter *Run an agent in this R session*

Description

peter() is the single entry point. Give it a quoted prompt and, optionally, the objects the agent should work on; it returns the agent session, which you can print, query ($text, $value, $usage) and continue with the pipe. Models, modes, skills, plugins, extensions and tools may be written as bare names.

Usage

```
peter(..., model = NULL, mode = NULL, skills = NULL, plugins = NULL,
      extensions = NULL, tools = NULL, agents = NULL, parallel = NULL,
      choices = NULL, levels = NULL, threshold = 0.5,
      min_confidence = NULL, uncertain = NULL,
      prompt = NULL, envir = parent.frame(), background = FALSE,
      budget = NULL, replay = NULL, .opts = list(), .run = TRUE,
      .stdin = FALSE)
```

Arguments

...	A session to continue, the prompt, and context objects.
model	A model name (sonnet, "anthropic/claude-sonnet-5-5"), a provider or router spec such as gptr_fake_provider(), or NULL for the configured default. A call made by the agent's own R code, and a team or fan-out it starts, uses the running session's model instead.
mode	One of plan, manual, edits, auto, yolo; NULL for the configured default. The plan a plan run proposes goes once, as a <plan> block, to the next top-level peter() call from the same environment, within an hour and in another mode, whatever that call asks: a script runs the plan with peter("Go ahead.", mode = auto). Another peter() call first, or a next call inside a loop or a run, discards it; options(gptr.plan_handoff = FALSE) turns the hand-off off. Interactively a plan run then asks whether to execute the plan: a mode chosen there is the session's, and its next prompt, such as Go ahead., runs the plan.
skills, plugins, extensions	Names (bare or quoted) of skills to preload, plugins and extensions to enable for this session; extensions also takes function(gptr) factories and file paths.
tools	Tool names, "+name"/"-name" modifiers, one preset name, or gptr_tool() specs. A bare tool name adds the tool, like "+name"; to narrow the set, choose a preset such as readonly or remove tools with "-name". A name that is neither a tool nor a preset is refused.
agents	A named list of agent definitions for a team (agent() means gptr_agent()). A decision model refuses it and parallel: it answers in one call.

`parallel` Fan one list-like context object out over this many concurrent sub-agents. Each child gets its element alone as `.x`, a named list of one attached as `.x[["A"]]`; `.opts = list(context = "none")`, which would drop it, is refused. A child that succeeds keeps only the values it returned with `gptr_return()`, so they must be self-contained.

`choices, levels, threshold`
System 1 answer shape, for a decision model (`jev`, `"ollama/clef-flash"`; after `gptr_config(system1 = "emulate:<ref>")`, `model = jev` runs it through a chat model). A conversational model refuses them, `min_confidence` and `uncertain` (`threshold` only when it is not 0.5) rather than ignore them. `choices` given as a factor, or `.opts = list(output = "factor")`, return a plain factor with those levels and no probabilities or confidence.

`min_confidence` A number from 0 to 1. An answer whose confidence is below it is uncertain, as is an unknown confidence unless it is 0. A decision's confidence is `abs(2 * p - 1)`, so 0.8 keeps only P(yes) at most 0.1 or at least 0.9. For choices and scores it is the engine's own scale, so a value tuned on Jev abstains differently on Clef (see `gptr_prob()`).

`uncertain` What an uncertain answer becomes: `NA` (the default), `TRUE` or `FALSE` (yes/no only), `function(state, answer)` returning one replacement of the answer's type, or `"stop"` for a `gptr_error_s1_uncertain` error. It has no effect without `min_confidence`.

`prompt` An explicit prompt; wins over positional selection.

`envir` Where the agent evaluates R code and creates objects. A call made by the agent's own R code without `envir` evaluates in a new environment whose parent is the caller's, `res$envir`: it reads the caller's objects and keeps the ones it creates there. A function frame (the default inside a function, or `environment()`) serves only that call, so the session keeps no workspace for later turns; pass `envir = new.env()` to keep one.

`background` Experimental: return the running session at once (needs later).

`budget` `list(tokens =, cost =, turns =)` limits for this call. `turns` counts model requests, retries and sub-agents' requests included; `session$turns` counts prompts instead. `cost` counts priced requests only: a model without prices (such as Codex, or a `gptr_provider()` without `prices`) has an unknown cost that counts nothing, so the call says once that its cost limit is not enforced; the `tokens` and `turns` limits still apply.

`replay` Document replay mode: `"auto"`, `"replay"`, `"live"` or `"record"`.

`.opts` Rare switches such as `thinking`, `max_turns`, `context`, `returns`, `images`, `preset`, and entries named by a plugin namespace. `thinking` applies to this call and to the session's later turns; `max_turns` caps this call's model requests.

`.run` `FALSE` builds the session with the prompt queued and returns it (see `gptr_step()`). Building checks egress and replay only; `gptr_step()` and `gptr_wait()` check the model's provider and key before they start the run. A piped `.run = FALSE` call applies its model, mode and tools at once; a `gptr_step()` or `gptr_wait()` that then refuses leaves them, with no turn.

`.stdin` Drive the console from piped standard input.

Details

Conversational agents compute in memory: their R tools use the objects already loaded in `envir`, and later turns can reuse fitted models and intermediate results without saving and reloading the dataset. R computes on the full objects, while the model receives size-limited descriptions, previews and tool output that may include data values. Ordinary R allocations, file-backed inputs, checkpoints and worker transfers still have their usual costs. See `vignette("execution-model", package = "gptr")` for these boundaries.

Agent code runs in this R process: a program it starts with `system()` or `system2()` reads standard input from your terminal and prints there, out of the agent's sight, so the agent is told, except under the `"minimal"` preset that sub-agents run, to use `peter$sh()` instead.

The dots take, in order: at most one leading session to continue, the prompt (the first unnamed string literal, else the first unnamed length-1 character value), and context objects (unnamed ones are labeled by their expression, named ones by name). Conversational context bindings refer to objects where they live; descriptions are budgeted separately from in-memory R computation. Every other argument is matched by its exact name only: a named object whose name abbreviates one and whose value fits it, such as `mo = "plan"`, stays context and warns (`gptr_warning_partial_name`).

A literal prompt gets `{name}` interpolation from `envir` (atomic vectors of 1 to 50 values; `{{` and `}}` are literal braces); `.opts = list(interpolate = FALSE)` switches it off.

Terminal statuses become conditions carrying the session as `$session`: `error` signals `gptr_error_provider`, `blocked` `gptr_error_permission` (`gptr_error_noninteractive` when the `ask` tool has no one to answer), `budget` `gptr_error_budget_<kind>` and `max_turns` `gptr_error_max_turns`. A team (`agents =`) or fan-out (`parallel =`) takes the status of its first member that did not end idle, with `$reason` naming each such member; when no member ended idle it signals that member's condition with the team as `$session`. A call whose model cannot be reached (no egress acknowledgment, replay mode, a disabled provider, a missing key) is refused before its run: no turn, no model or mode switch, no new `gptr_last()` and no `$session`. So is a call with an unknown plugin or `skills =` name, `.opts$images` its model does not take, or a secret-looking prompt you decline to send. A piped call loads its `extensions`, `plugins` and specs first, since they can define that model, and unloads them again when it is refused or interrupted before its turn starts; registry filters (`plugins = "-x"`) stay on the R session only once a call is accepted. `peter()` with no prompt opens the console when someone can answer, and signals `gptr_error_noninteractive` otherwise.

A new interactive console without an explicit or configured model first offers CLI selection or manual API/Ollama instructions. Selecting an available CLI saves its default at user scope; cancellation (choose Cancel or press Enter) or manual setup returns without starting a session. Existing sessions, explicit or configured models and `.stdin = TRUE` skip setup. CLI-reported login is not proof of online access or subscription billing. See `vignette("interactive-console", package = "gptr")`.

The name honors Peter Naur's work on readable language definitions and Peter Cathcart Wason's research with Jonathan Evans on reasoning. In Naur's spirit, gptr records sessions as runnable documents. See `vignette("why-peter", package = "gptr")` for the history and sources. A user object named `peter` hides the gateway; call `gptr::peter()` then.

Value

A `gptr_session` for agent work (invisibly when its answer was streamed to the console), or a typed System 1 vector for classifier models.

The interactive console

`peter()` without a prompt opens a console when someone can answer (interactive R or IRkernel); `peter(.stdin = TRUE)` reads its input from standard input, and `session |> peter()` opens it on `session`. With no explicit or configured model, the interactive console first offers Codex CLI, Claude Code CLI or API configuration instructions. Choosing an available CLI saves its default model at user scope; cancelling returns to R. Piped input skips this setup. Each input is one of:

- a prompt, sent as `session |> peter("...")` with `{name}` interpolation; `@file` adds the head of a file (at most 40 lines; the files of one prompt share about 2,000 tokens) and `@object` attaches an object by name;
- `!code`: R code run in the session's environment; the code and its output go with the next prompt (at most 300 estimated tokens). `!!code` omits that note. Both execute your own R immediately, without the model's permission gate or a model request. Workspace changes remain visible to later calls, and enabled transcripts can record the code; `!!` is not a privacy boundary. A fenced block marked `r`, `R`, `{r}` or unmarked works like `!code` (other fences are prompt text), a `"""` block is a multi-line prompt, and a trailing backslash continues a line;
- a slash command such as `/model`, `/mode`, `/cost` or `/exit`; `/help` lists them all.

In CRAN's R for macOS in a terminal a pasted block is one input: its lines join the first one and `...` waits for Enter (text typed there is added); a pasted fence or `"""` block is read up to its closer at once, a block pasted at an approval or a choice is kept in the console history and the question asked again, and one pasted at the pause menu's `? Steer` or `? Follow-up` is the whole text. A last pasted line without a newline waits for Enter at the next prompt, so a two-line paste without a final newline answers a question with its first line. Elsewhere (Homebrew or conda R, Linux terminals, RStudio, R.app, Positron, Windows) pasted lines arrive one at a time, as typed lines do (at a question the first one answers it); wrap a multi-line paste in `"""`. Lines typed, pasted or run from a script before a question (an approval, a choice) appeared never answer it: while an answer runs they are queued for the agent as `type_ahead` says (below; a /command or !code line waits for the next prompt), otherwise they go to the console history. Answer after the prompt.

What you type while peter works, and the key that pauses it, depend on the console:

Console	Typed while peter works	Pause
Terminal R on macOS or Linux, an IDE's Terminal tab	a draft: Enter steers, Tab sends a follow-up	Ctrl-C
TERM=dumb, Emacs shell, ESS	each line, as `type_ahead` says	Ctrl-C
RStudio, R.app	the next prompt, unless a question reads it	Esc
Positron	R code, run after `peter()` returns	Ctrl-C
Windows Rterm	the next prompt, unless a question reads it	Ctrl-C
Windows Rgui, Jupyter	the next input	none: Esc or interrupt
Rscript, pipes, knitr	nothing; `.stdin = TRUE` reads the pipe	none

In terminal R and an IDE's Terminal tab what you type while peter works is a draft on the bottom row, under the transcript, so it never mixes into the answer; a block pasted there is one message where the terminal supports bracketed paste. Enter sends it as a steer: the agent reads it at its next step boundary, when the response in flight ends after the results of all its tool calls (an answer about to end gets one more request), and it then shows as your own line. Tab sends it as a follow-up: the next prompt, sent as its own run once the answer ends. `type_ahead = "follow_up"` (`gptr_config()`) swaps the two keys. A draft that starts with `/` or `!` is always a follow-up, so `/command` and `!code` lines run at the next prompt, in the order typed. The bottom row counts what

is queued; Up on an empty draft takes back the newest message not yet delivered, to edit and send again, and Ctrl-U clears the draft. A question (an approval, a choice) waits until typing pauses for a second, or three while the draft has text, and the draft stays: through questions, Ctrl-C and the pause menu, and to the next prompt, where Enter sends it. Keys typed while R code runs in this process (the `r` tool) show when it returns. Such code that reads the terminal through `readline()` or `menu()` inside another function shows only that read's prompt (the text printed before it shows when the code returns, the typed answer never), and Ctrl-C at that read aborts the answer instead of pausing it. A steer the answer did not take (it ended first) is the next prompt; after an abort or an error nothing queued is sent: the queued text and the draft go to the console history, and Up recalls them. Under TERM=dumb or in ESS a line typed with Enter is queued as `type_ahead` says. On the CLI routes (`claude-cli`, `codex`) the agent reads a steer after the CLI's answer, which is often the whole run.

RStudio, R.app, Positron and Windows consoles hold what is typed until R reads again. In RStudio, R.app and Windows consoles a line typed while peter works is the next prompt, one prompt per line, unless a question comes first: it reads the line, queues it as `type_ahead` says and asks again. Positron's console keeps typed text as R code, which runs after `peter()` returns, and clears it at Ctrl-C. To steer in these consoles, pause and press `s` (Windows' Rgui cannot pause). In RStudio an Esc first closes RStudio's own completion list, if open, then clears a line being typed, so the menu comes at the next Esc. During `peter("...")` at R's prompt, outside the chat, typed lines stay R's next input.

Tab completes only after `@` (objects, then files), `@obj$` (its members), `peter$`, a leading `/` (commands) and on a `!` or `!!` line (R code), never in plain text: in terminal R, R in an IDE's Terminal tab, Windows' Rgui and Rterm, and RStudio's console, which completes only what its popup inserts as offered (the commands after `/` and R code after `!`, not words holding `@` or `$`, which it would wrap in backticks). In a terminal Tab completes the draft the same way while peter works, and on such a word never sends it. R.app, radian and Positron's console keep their own completion. gptr never changes RStudio's preferences: as you type in its console, RStudio adds closing brackets and quotes unless `insert_matching` is off, and opens completions unless `console_code_completion` is off (Tools > Global Options > Code).

The pause key opens a pause menu while an answer runs, with the keys `s` steer, `f` follow-up, `c` continue, `a` abort and, for a foreground call (a chat turn included) when available, `b` background. `s` and `f` ask for the text and queue it as Enter and Tab do; in a `peter("...")` call a follow-up is sent when the agent would otherwise stop. A key typed right after the pause key answers the menu. Continuing leaves the work as if it had not paused, and pressing the pause key again aborts. Abort keeps the partial session; what you queued and the draft go to the history, unsent. Background returns to the chat prompt and keeps the run: what was queued for after the answer and the draft go to the history, unsent, as after an abort, and another prompt steers the running session. Nothing runs it while the chat waits for input: use `!!gptr::gptr_wait(gptr::gptr_last())` there to continue it, its answer then shown as a prompt's and Enter steering it. Under Rscript, background runs advance only inside blocking gptr calls. Where an interrupt cannot be resumed (Windows' Rgui, Jupyter) the key stops the answer instead.
`/exit`, two consecutive interrupts at the prompt or the end of piped input return the session invisibly, so `session = peter()` keeps the conversation. In a terminal the chat reads keys with the terminal's echo off and restores it on return, on an error, at an interrupt and at `q()`; if R is killed or crashes while peter works, type `reset` and press Enter to get the terminal back.

Options: `gptr.verbose` (0 silent, 1 progress, 2 streamed console, 3 debug; default 0 in knitr and testthat, 1 under Rscript, 2 at the console), `gptr.max_turns_console` (turns per console prompt,

default 200) and `gptr.history` (console inputs to R's history, default `TRUE`). Ordinary progress uses normal output; actual errors use stderr. `vignette("interactive-console", package = "gptr")` gives input examples and the complete built-in command table.

Sessions and results

A pipe such as `session |> peter("Explain the result.")` adds work to the same session and kept workspace. Use `gptr_fork()` to branch explicitly. The last turn's answer is `session$text`; `session$value` is the newest R value selected by `gptr_return()` in any turn, so a turn that selects none keeps it. Agent-created objects remain in `envir`.

See Also

gptr_security, gptr_egress and gptr_options; `vignette("language-models", package = "gptr")` for connecting models and scripted calls.

Examples

```
fake = gptr_fake_provider(list("The data has 32 rows.", "The data has 11 columns."))
session = peter("How many rows does the data have?", mtcars, model = fake)
session$text
session |> peter("And how many columns?")
identical(gptr_last(), session)
```

---

print.gptr_session *Print a session*

Description

The footer is `status . model . turns . tokens . cost . id`. A token count or cost that a provider did not report is printed as unknown, never as zero. A session replayed from a document shows the status `replayed` until it makes a request; rebuilt from its saved transcript it shows the recorded usage, reconstructed from the document alone unknown usage. A team or fan-out without an answer of its own prints each child's answer under a `### <name> (<model>)` heading, with `(ended with status <status>)` for a child that did not finish.

Usage

```
## S3 method for class 'gptr_session'
print(x, ...)
```

Arguments

`x`	A `gptr_session`.
`...`	Unused.

Value

`x`, invisibly.

print.gptr_session_summary *Print a session summary*

Description

Display the latest recorded messages at the current console width. This changes neither the complete history table nor the session, and makes no model request. Unknown token counts remain unknown. Previews are single lines; the underlying table retains its text.

Usage

```
## S3 method for class 'gptr_session_summary'
print(x, n = 8L, ...)
```

Arguments

`x`	A `gptr_session_summary`.
`n`	Maximum number of recent messages to display; `Inf` displays every row.
`...`	Unused.

Value

`x`, invisibly.

str.gptr_session *Show a session's structure*

Description

Show a session's structure

Usage

```
## S3 method for class 'gptr_session'
str(object, ...)
```

Arguments

`object`	A `gptr_session`.
`...`	Unused.

Value

`NULL`, invisibly.

summary.gptr_session *Summarize a session as its history*

Description

Reads a table of recorded turns without a model request. It does not generate a narrative workflow summary or export an executable script. To request prose, continue the session with peter() and verify the model-generated answer against the recorded work. Both summary(session) and session$history retain every recorded message. Printing shows the latest eight messages and a snapshot of session metadata. Notebook, R Markdown and Quarto displays use the same compact preview by default. Use print(x, n = Inf) to display every row, or as.data.frame(x) for an ordinary data frame without the display metadata.

Usage

```
## S3 method for class 'gptr_session'
summary(object, ...)
```

Arguments

object	A gptr_session.
...	Unused.

Value

A gptr_session_summary data frame (turn, role, preview, tools, tokens).

Index

all.names(), *23*
as.character.gptr_session
(format.gptr_session), 5
as.data.frame.gptr_session_summary, 4

background-sessions (gptr-background), 6

do.call(), *74*

format.gptr_risk, 4
format.gptr_session, 5

gptr-background, 6, *73*, *80*
gptr_adapter, 7
gptr_agent, 8, *70*
gptr_agent(), *81*
gptr_agents (gptr_skills), 69
gptr_api, 9
gptr_api(), *72*
gptr_artifacts, 10
gptr_backend, 11
gptr_blocks, 12
gptr_blocks(), *27*, *71*
gptr_cache, 12
gptr_cancel, *6, 7*, 13
gptr_cancel(), *73*, *80*
gptr_check, 14, *46*, *77*
gptr_checkpoints (gptr_rewind), 57
gptr_coach, 15
gptr_coach(), *24*, *66*, *68*, *75*
gptr_command, 17
gptr_config, 17
gptr_config(), *25*, *36*, *38*, *42*, *84*
gptr_context_block, 19
gptr_describe, 20
gptr_doc, 21
gptr_doc(), *27*, *43*, *63*, *70, 71*
gptr_egress, 23, *42*, *49*, *63*, *68*, *86*
gptr_env, 25
gptr_env(), *42*, *67*
gptr_export, 26
gptr_export(), *63*
gptr_fake_provider, 28
gptr_fake_provider(), *25*, *81*
gptr_fork, 29
gptr_fork(), *58*, *86*
gptr_hook, 30
gptr_init, 30
gptr_init(), *18*, *63*, *66, 67*
gptr_jobs, *6, 7*, 31
gptr_last (gptr_sessions), 68
gptr_last(), *35*, *74*, *83*
gptr_login, 32, *33*
gptr_logout (gptr_login), 32
gptr_mcp, 33
gptr_mcp_add, *33*, 34
gptr_mcp_remove, *33*
gptr_mcp_remove (gptr_mcp_add), 34
gptr_mcp_serve, *33*, 35
gptr_mcp_serve(), *42*, *68*
gptr_models, 36
gptr_on, 37
gptr_options, *18*, *25*, 38, *64*, *68*, *86*
gptr_parallel, 42
gptr_parallel(), *39*
gptr_permissions, 44
gptr_permissions(), *18*, *68*, *74*
gptr_plugins, 45, *70*
gptr_policy, 46
gptr_preimage, 47
gptr_prob, 48
gptr_prob(), *82*
gptr_prompt, 49
gptr_prompt_section, 50
gptr_provider, 51
gptr_provider(), *82*
gptr_providers, 52
gptr_providers(), *25*, *37*
gptr_redact, 53

gptr_register, 55, *77*
gptr_registry, *46*, 55
gptr_reload, *46*, 56
gptr_resume *(*gptr_sessions*)*, 68
gptr_return, 57
gptr_return(), *82*, *86*
gptr_rewind, 57
gptr_rewind(), *66*
gptr_risk, 60
gptr_risk(), *44*, *64*, *68*
gptr_router, 61
gptr_scrub, 62
gptr_scrub(), *63*, *67, 68*
gptr_security, *25*, *40*, *42*, *44*, *59*, 63, *86*
gptr_sessions, 68
gptr_skills, 69
gptr_source, 70
gptr_source(), *21*, *27*, *73*
gptr_spec, 72
gptr_spec(), *55*
gptr_steer, *7*, 72
gptr_steer(), *73*
gptr_step, 73
gptr_step(), *73*, *82*
gptr_tailor, 74
gptr_tailor(), *15, 16*
gptr_tool, 76
gptr_tool(), *81*
gptr_tool_result, 77
gptr_tool_result(), *76*
gptr_trust, *70*, 78
gptr_trust(), *18*, *30*, *66*, *68*
gptr_usage, 79
gptr_wait, *6, 7*, 80
gptr_wait(), *73*, *82*

lapply(), *74*

peter, *7*, 81
peter(), *15, 16*, *18*, *24*, *37–39*, *42, 43*, *63*, *74, 75*, *88*
print.gptr_prompt_view *(*gptr_prompt*)*, 49
print.gptr_risk *(*format.gptr_risk*)*, 4
print.gptr_session, 86
print.gptr_session_summary, 87

saveRDS(), *75*
source(), *70*
str.gptr_session, 87
summary.gptr_session, 88
Sys.setenv(), *25*