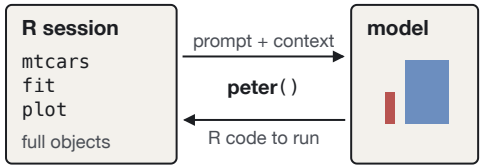
Agents in your R session with gptr :: CHEATSHEET

`peter()` runs language model agents in your current R session, on objects already in memory. 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.

How it works



R uses full objects; models see descriptions, previews and tool output, including data values (`?gptr_egress`).

gptr_describe(`mtcars`)
What a model is shown of an object.

```
peter("Say hello.",
  .opts = list(context = "none"))
```
Workspace/object descriptions off; prompts, instructions and environment still go. Remote consent still applies.

Set up

OpenAI for live calls; Try it offline needs no key and defines `fake` and `s` for page 2.

```
pak::pak("Broccolito/gptr@main")
library(gptr)
```
R 4.2+; first install pak with `install.packages("pak")` if needed.

gptr_login(`"openai"`)
Interactive R: enter your OpenAI key, masked.

gptr_providers(); **gptr_models**(`"gpt"`)
List the routes; search the offline model catalog.

```
gptr_config(model = "gpt",
  .scope = "user")
```
Save a default model (session, project or user).

gptr_init(`"."`)
Create `.gptr/`; keep sessions and records.

gptr_trust(`".", trust = TRUE`)
Trust the project's settings, skills and plugins.

```
e = gptr_config()$egress
e$openai = "ack"
gptr_config(egress = e, .scope = "user")
```
Consent to OpenAI context in scripts.

Try it offline

```
step = list(tool = "r", input = list(
  code = "fit = lm(mpg ~ wt, mtcars)"))
fake = gptr_fake_provider(
  list(step, "Done."))
s = peter("Fit mpg on wt.", mtcars,
  model = fake, mode = "auto"); coef(fit)
```
After `library(gptr)`: no key or network.
The scripted replies fit the model, then set `s$text` to "Done."

Ask Peter from R

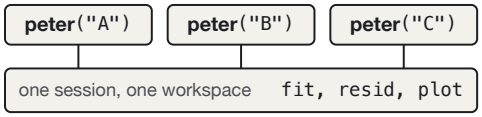
Live calls follow Set up. `mtcars` is built in; quoted model names are strings. In functions use `envir = new.env()` to keep objects.

```
fit_s = peter("Fit mpg ~ wt as fit.",
  mtcars, mode = "auto")
```
Prompt, then data. `auto` permits ordinary changes in scripts; critical actions still ask.

fit_s$text; fit_s$value; fit_s$turns
Answer, newest `gptr_return()` value (else NULL), prompt count.

```
peter("Summarize.", mtcars,
  budget = list(tokens = 5e4, turns = 5))
```
Per-call limits; `turns` counts model requests.

Pipe: one session, many turns




```
fit_s |> peter("Plot the residuals.")
```
Continue the live fit above, with its workspace.

```
fit_s |> peter("Check the fit.",
  model = "gpt:high")
```
Change reasoning; keep the history.

```
b = gptr_fork(fit_s)
b |> peter("Try a log model.")
```
Branch; reference-object edits can reach the original.

```
fit_s; summary(fit_s); gptr_last()
```
Answer, history, most recent session.

The console

```
gptr_doc("log.R")
session = peter()
session |> peter()
```
Interactive R: bind a transcript, then chat. `/exit` returns the session; the next call reopens it.

INPUT SYNTAX AT `peter>`

`Fit mpg on wt.`	A prompt for the session
`!head(mtcars)`	Run R and show the model
`!!x = 2`	Run R; send no code or output
`` ```r … ``` ````	Fenced R, run like `!code`
`""" … """`	A multi-line prompt
`line \`	Continue on the next line
`@notes.md`	Your existing file: first 40 lines
`@mtcars`	Attach an object
`{pi}`	Insert an R value

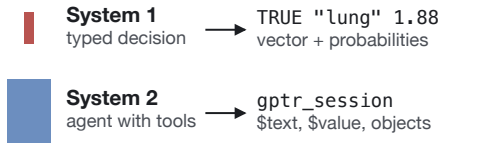
SLASH COMMANDS: ONE AT A TIME

`/help [cmd]`	Commands and syntax
`/model /mode`	Show or switch either
`/status`	Session, model, mode
`/cost /env`	Tokens and cost; objects
`/tools`	Tools and helpers
`/compact`	Compact the history
`/clear`	New chat, same objects
`/resume /fork`	Resume or fork a session
`/doc [path]`	The history document
`/skills /mcp`	List skills, MCP servers
`/<name>`	A `.gptr/prompts/` template
`/permissions`	Show or change rules
`/undo /redo`	Undo or redo a turn
`/retry`	Resend the last prompt
`/rewind [k]`	Keep turns 1 to k
`/exit`	Leave; return the session

WHILE AN ANSWER RUNS

In a terminal (not Windows) type, then Enter steers and Tab follows up. Ctrl-C (RStudio: Esc) pauses: **[s]**teer, **[f]**ollow-up, **[c]**ontinue, **[a]**bort, **[b]**ackground. Approvals: **[y]**es, **[a]**lways (session), **[p]** always (project), **[n]**o (`n <text>` says what to do instead).

System 1 and System 2



Jev and Clef return one typed decision per input. Conversational models work in turns with tools and return a session.

SYSTEM 1: TYPED DECISIONS

```
gptr_login("typesafe")  # interactive R
e = gptr_config()$egress
e$typesafe = "ack"
gptr_config(egress = e, .scope = "user")
```
Jev uses its own TypeSafe key and consent.

```
abstracts = c("Randomized trial.",
  "Observational study.")
rct = peter("Is this an RCT?", abstracts,
  model = "jev")
```
Logical: TRUE when P(yes) ≥ 0.5 (the default threshold).

```
gptr_prob(rct)
abstracts[rct]; sum(rct)
```
P(yes); subset and count these typed answers.

```
peter("Which tissue?", "Hepatocyte",
  model = "jev",
  choices = c("liver", "lung", "other"))
```
One label per input.

```
peter("How severe?",
  "Pain limits walking.", model = "jev",
  levels = c("low", "mid", "high"))
```
A fractional score: the expected 0-based level.

```
peter("Is this an RCT?", abstracts,
  model = "jev", min_confidence = 0.8)
```
Below the minimum (|2p−1| for yes/no), return NA or what `uncertain` says.

```
fit_s |> peter("Did it succeed?",
  model = "jev")
```
Judge the live fit's last answer; add no turn.

SYSTEM 2: AGENTS

```
peter("Plan an mpg analysis.", mtcars,
  model = "gpt",
  .opts = list(thinking = "high"))
```
Reasoning depends on the model; a suffix such as `gpt:high` works too.

Models and providers

```
"anthropic/claude-sonnet-5-5:high"
 └──────┘ └──────────────────┘ └──┘
 provider       model id      thinking
```

Aliases: `"sonnet"`, `"opus"`, `"haiku"`, `"gpt"`, `"flash"`, `"jev"`, `"claude_code"`, `"codex"`.

`anthropic/…`	`ANTHROPIC_API_KEY`
`openai/…`	`OPENAI_API_KEY`
`google/…`	`GEMINI_API_KEY`
`openrouter/…`	`OPENROUTER_API_KEY`
`azure/…`	`AZURE_OPENAI_API_KEY`, `AZURE_OPENAI_ENDPOINT`
`bedrock/…`	`AWS_BEARER_TOKEN_BEDROCK`, `AWS_REGION`
`ollama/…`	Local server; also lmstudio, vllm
`claude-cli`	Shell: `claude auth login`
`codex`	Shell: `codex login`
`jev`	`TYPESAFE_API_KEY`
`ollama/clef`	Installed model; Ollama 0.35.1+

```
gptr_models(provider = "ollama",
  refresh = TRUE)
```
Installed models on a running Ollama server.

```
gptr_register(gptr_provider("corp",
  "openai-completions",
  base_url = "https://HOST/v1",
  auth = "CORP_KEY", models = list(
    list(id = "MODEL",
      tool_call = TRUE))))
```
Template: replace HOST/MODEL; set CORP_KEY for calls. Same-name registration replaces it.

Permissions and trust

`plan`	Read only; next top-level call gets the plan
`manual`	Reads run; classified changes ask
`edits`	Also allows file edits in the project
`auto`	Critical actions and secret reads ask
`yolo`	Broad; critical and secret-flow guards remain

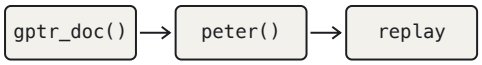
```
gptr_permissions(deny = "r(sh:rm *)")
```
Rules: deny beats ask beats allow.

```
gptr_rewind(s)
```
Page 1's s. Restore checkpoints; not references or external effects. Limits: `?gptr_rewind`.

Auto/yolo: computed `system()` needs an explicit `r(fn:system)` allow rule. Advisory, not a sandbox: `?gptr_security`.

Record, replay and export

R console, in order: use a new writable project and page 1's fake. Leave GPTR_REPLAY unset for this first recording.




```
gptr_register(
  gptr_fake_provider(fake$script))
```
Reset/register the offline model.

```
writeLines("library(gptr)", "analysis.R")
gptr_doc("analysis.R")
s = peter("Fit mpg on wt.", mtcars,
  model = "fake/fake-1", mode = "auto")
gptr_doc(FALSE)
```
Create, bind, record a top-level call, then unbind.

```
gptr_source("analysis.R",
  replay = "replay")
gptr_blocks("analysis.R")
```
Replay; list fresh/stale/edited/undone blocks.

```
GPTR_REPLAY=replay Rscript analysis.R
```
POSIX shell (macOS/Linux), after recording. No model calls.

```
gptr_export("analysis.R", "plain.R")
```
Code runs; peter calls become comments. Needs fresh blocks and literal arguments.

```
gptr_export("analysis.R", "peter.R",
  format = "peter")
```
Calls run; code is commented. Live runs need the provider.

```
gptr_doc("report.Rmd")
```
Existing report: bind in setup before calls; rmarkdown/Pandoc renders it. Quarto needs its CLI.

```
gptr_doc("analysis.ipynb", sync = TRUE)
```
Existing recorded notebook: close before syncing.

Replay needs the code's inputs/packages; cached text needs .gptr/. Top-level `gptr_coach()` replays too.

Learn more

Guides, Harnesses and coaching, and the function reference.

Tools, files and artifacts

In a new writable project. Presets: minimal, standard, readonly (r can write), extended; select with `tools = "extended"`.

```
dir.create("R", showWarnings = FALSE)
peter$write("R/fit.R", "n=10\n")
peter$read("R/fit.R", limit = 40)
peter$grep("n=", path = "R")
peter$find("*.R"); peter$ls("R")
```
Create the fixture; read, search and list.

```
peter$edit("R/fit.R", list(list(
  oldText = "n=10", newText = "n=20")))
peter$write("notes.txt", "done\n")
```
Edit the fixture; write files atomically.

```
peter$sh(c("Rscript", "--version"))
peter$bg(c("Rscript", "--version"))
```
Run/background a program; Rscript uses this R.

```
peter$sql("SELECT cyl, AVG(mpg)
  FROM mtcars GROUP BY cyl", mtcars)
```
Data frames need DBI + duckdb. `peter$py()`: reticulate + Python.

```
apps = gptr_artifacts()
if (nrow(apps))
  gptr_artifacts(apps$id[[1]], open = TRUE)
```
After asking Peter to build a Shiny app: use its actual ID. Shiny required; scripts do not launch by default.

Skills, plugins and MCP

```
gptr_skills(); gptr_plugins()
```
List available skills and plugins.

```
peter("Review lm().", stats::lm,
  skills = "high-performance-r")
```
Bundled skill; needs page 1 model setup.

```
peter("Report units.", plugins = "./lab")
```
Requires trusted ./lab with plugin.json.

```
gptr_mcp_add("docs",
  url = "http://127.0.0.1:8000/mcp")
```
Your server's URL (or command for stdio); connects on use.

```
peter$mcp[["SERVER"]][["TOOL"]]()
```
Template: replace names and tool arguments.

```
mcp = gptr_mcp_serve()
as.character(mcp$config$claude_code)
```
httpuv/later/openssl needed. Paste into Claude Code; keep R open. When done: `mcp$stop()`.

```
up = gptr_tool("shout", "Upper-case.",
  fun = function(x) toupper(x))
off = gptr_register(up)
```
Register a tool; off() removes it.

Teams, fan-outs, background

Needs page 1's model setup. Modes inherit; a team throws only if all children fail. Inspect statuses.

```
team = peter("Review mtcars.", mtcars,
  agents = list(
    stats = gptr_agent(system = "Stats"),
    code = gptr_agent(system = "Code")),
  mode = "auto")
```
Named roles; `team$stats$text` reads one.

```
fan = peter("Count rows; gptr_return().",
  list(mtcars, iris), parallel = 2,
  mode = "auto")
```
One child per element; values in fan$value. `gptr_parallel()`: unrecorded calls; use separate envir values.

```
bg = peter("Count rows.", mtcars,
  background = TRUE, mode = "auto")
gptr_wait(bg)
```
Experimental; wait finishes the run.

```
gptr_usage(team, by = "agent")
```
Usage by session/agent/model/route.

Tailor and coach a harness

Page 1 model setup required. Coaching runs unattended R processes; most tasks need auto.

```
h = gptr_tailor(peter, mode = "auto",
  coaching = list(note = "Give units."))
```
A call's arguments win; its tools add.

```
h = gptr_tailor(h, rules = list(never =
  list(list(rule = "r(fn:stepAIC)",
    why = "No stepwise selection."))))
```
Hard never rule; require requests a missing step.

```
h("Describe mtcars.", mtcars)$text
```
Call like peter, pipes included.

```
x = c(1, 2, 3)
q = "Mean x; gptr_return() it."
h = gptr_coach(q, x, harness = h,
  check=function(r) isTRUE(r$value==2))
```
Kept only if acceptance checks pass. Optional `coach = "claude_code"` needs its CLI login.

```
saveRDS(h, "h.rds")
h = gptr_tailor(readRDS("h.rds"))
```
Rebuild a saved harness; checks it and shows fixed code.

```
gptr_tailor(h, dir = "my-harness")
```
Write a new Markdown output folder; gptr never reads it back.