Package 'commons'

September 11, 2026

Title AI Agents for Data Analysis

Version 0.1.0

Description Implements trustworthy large language model agents.
Connect raw data sources, a pool of trusted calculations, and a searchable
context layer that demonstrates how to interpret them.
Then, deploy data agents that answer questions, log interactions, and
can be evaluated and improved over time.

License MIT + file LICENSE

URL https://github.com/posit-dev/commons,
https://posit-dev.github.io/commons/

BugReports https://github.com/posit-dev/commons/issues

Depends R (>= 4.1.0)

Imports bslib (>= 0.11.0), callr, cli, coro, DBI, duckdb (>= 1.5.4.2),
ellmer (>= 0.5.0), evaluate, filelock, highr, htmltools, httr2
(>= 1.1.0), jsonlite, knitr, later, magick, processx, promises
(>= 1.5.0), R6, ragg, ragnar, rlang (>= 1.2.0), roxygen2, S7,
sass, shinychat (>= 0.5.0), utils

Suggests bit64, bsicons, chromote, dbplyr, dplyr, ggplot2, gt, odbc,
otel (>= 0.2.0), otelsdk (>= 0.2.0), pins, plotly, pkgload,
readr, rmarkdown, rsconnect, shiny (>= 1.11.1), shinytest2,
testthat (>= 3.0.0), vitals, withr, yaml

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first run-r, citation-scan, citation-browser,
commons, data-source, pool, definitions, trajectory-review

Encoding UTF-8

Config/roxygen2/version 8.1.0

Config/Needs/website tidyverse/tidytemplate

NeedsCompilation yes

Author Simon Couch [aut, cre] (ORCID: <https://orcid.org/0000-0001-5676-5107>),
Sara Altman [aut],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Simon Couch <simon.couch@posit.co>

Repository CRAN

Date/Publication 2026-09-11 15:30:15 UTC

Contents

`commons` *Create a commons agent*

Description

`commons()` creates an ellmer::Chat subclass with tools and prompting that allow the agent to navigate its data sources, semantic layer, and context layer. Depending on the agent's choice of tools, responses can be deterministically classified as based on a trusted calculation, cited, or untrusted.

Usage

```
commons(
  client,
  data_sources,
  semantic_layer = NULL,
  context_layer = NULL,
  ...,
  instructions = NULL,
  network = c("none", "full"),
  log = FALSE,
  share_with = NULL
)
```

Arguments

client	An ellmer::Chat giving the provider and model to use, e.g. `ellmer::chat_anthropic()`. For best results, enable thinking when supported by the selected provider and model. A system prompt already set on the client is ignored, with a warning; use `instructions` to add to commons' prompt.
data_sources	A `data_source()`, or a named list of them. Measures can take a source's connection as an argument named after the source; see `semantic_layer()`.
semantic_layer	An optional `semantic_layer()`.
context_layer	An optional `context_layer()`.
...	These dots are for future extensions and must be empty.

instructions	Optional instructions placed under an `## Additional instructions` heading at the end of commons' built-in system prompt, as a single string or the path to a text or Markdown file.
	`commons(`
	`  # ...`
	`  instructions = "Use the organization's fiscal-year conventions."`
	`)`

network	Whether the agent's R session has network access. One of `"none"` (the default) or `"full"`. The session uses OS sandboxing on Linux and macOS. On unsupported hosts, local development can opt in to best-effort R guardrails with `options(commons.allow_unsafe_fallback = TRUE)`. These guardrails are not a security boundary.
log	Whether to request conversation trajectory capture with OpenTelemetry (default `FALSE`). When `TRUE`, commons checks the tracing setup and warns with setup steps when it is incomplete. This feature requires Connect >= 2026.09.0. When deploying to Posit Connect, include `"OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTEN` in the `envVars` argument to `rsconnect::deployApp()` so **ellmer** includes message content. A server administrator must also set `OpenTelemetry.Enabled = true` and `OpenTelemetry.AllowContentInstrumentation = true`. Once the agent is deployed and serving traffic, you can read conversation histories back into R with `trajectory_read()`.
share_with	An optional character vector of Connect usernames granted access to this content's trajectories when running on Posit Connect. Reading traces requires editor-level access, so named users are added as collaborators on the content. Note that users whose Connect *account* role is viewer cannot read traces even when named here; trace readers need at least a publisher account.

Details

The provider and model come from `client`; commons sets its own system prompt and tools. Use `agent$chat()` to ask questions, `commons_theme()` and `commons_server()` to embed the agent in Shiny, and `vitals::generate()` to use the agent as a vitals solver.

Value

An ellmer::Chat subclass.

Cache pre-warming

A commons agent builds its context search index and downloads uncached pins the first time it needs them. `commons_server()` and `commons_app()` call the agent's `prewarm()` method automatically during post-startup idle time.

To warm the caches before deployment, call `agent$prewarm()` in a pre-deploy script. The context index is cached on disk once per version of the context documents; pin downloads populate the local pins cache.

To ship a pre-built context index with an app, configure a directory inside the app in both the pre-deploy script and the deployed app, then prewarm the agent before deploying:

```
options(commons.context_cache = "commons-cache")
agent <- commons(
  ellmer::chat_anthropic(),
  data_sources = data_source(sales = sales)
)
agent$prewarm()
```

Do not use `app_cache/` for this workflow because rsconnect excludes it from deployed bundles. Without explicit configuration, commons uses Connect's persistent content data directory when available, an `app_cache/` directory beside hosted apps, or the per-user cache directory. Set the cache directory with `options(commons.context_cache = "path/to/dir")` or the `COMMONS_CONTEXT_CACHE` environment variable. Set the option to `FALSE` to disable persistence. The cache is capped at 256 MB with least-recently-used eviction; change the cap with `options(commons.context_cache_max_size)`.

Agent tools

Depending on its semantic layer, context layer, and data sources, a commons agent receives some combination of these tools:

- `search_pool` searches trusted calculations and semantic models.
- `search_catalog` searches a warehouse catalog.
- `call_measure` invokes an R measure.
- `call_metrics` invokes governed or warehouse-native metrics.
- `call_calculation` invokes an exact trusted query.
- `search_context` retrieves relevant business context.
- `describe_table` inspects a table or semantic model.
- `run_sql` executes a read-only SQL query.
- `run_r` executes R code to analyze results and render plots in the agent's R session.

These model-facing tools should be considered private. Their constructors are intentionally not exported, and their names, arguments, availability, and behavior may change without notice. Application code should configure an agent through `commons()` and its layer constructors rather than depend on individual tools.

Examples

```
## Not run:
# A measure over local data computes directly in R.
sem <- semantic_layer(
  measure(
    "order_count",
    "Count of orders.",
    function() nrow(my_sales),
    arguments = list()
  )
)
agent <- commons(
  ellmer::chat_anthropic(),
  data_sources = data_source(sales = my_sales),
  semantic_layer = sem
)
agent$chat("How many orders are there?")

# A measure takes a connection as an argument named after a data source.
# `warehouse` isn't in `arguments`, so the model never sees it; commons
# supplies it when the measure runs. Bind model-supplied arguments through
# DBI so they're quoted safely.
con <- DBI::dbConnect(duckdb::duckdb())
sem <- semantic_layer(
  measure(
    "revenue_by_region",
    "Total revenue for a region.",
    function(region, warehouse) {
      DBI::dbGetQuery(
        warehouse,
        "SELECT sum(revenue) AS revenue FROM sales WHERE region = ?",
        params = list(region)
      )
    },
    arguments = list(region = ellmer::type_string("Sales region."))
  )
)
agent <- commons(
  ellmer::chat_anthropic(),
  data_sources = list(warehouse = data_source(con)),
  semantic_layer = sem
)

# Objects that aren't data sources (a pins board, an API client) come from
# argument defaults in the measure, e.g. `board = pins::board_connect()`.
# See ?semantic_layer.

## End(Not run)
```

commons_app *Shiny chat app for a commons agent*

Description

commons_app() composes commons_server() and commons_theme() into a complete app for local development. To customize and deploy the Shiny app, assemble the UI and server yourself with commons_theme() and commons_server().

Usage

```
commons_app(client, ...)
```

Arguments

client	A commons() agent.
...	Extra arguments passed to shiny::shinyApp().

Value

A shiny::shinyApp() object.

Citations and provenance

The server verifies citations against trusted calculations, context, and data documentation as the answer streams. Verified citations appear inline, with details that name the trusted source. A provenance marker follows the answer when it was produced by a trusted calculation, or when a fallback answer cites nothing verified.

Examples

```
## Not run:
agent <- commons(
  ellmer::chat_anthropic(),
  data_sources = data_source(sales = sales)
)
commons_app(agent)

## End(Not run)
```

commons_server *Chat server and theme for custom commons apps*

Description

These are the building blocks for deploying a commons chat as a Shiny app; for local development, use `commons_app()`. Pair `commons_server()` with `shinychat::page_chat()` or `shinychat::chat_ui()`, passing `theme = commons_theme()` so the commons chat assets are on the page.

Usage

```
commons_server(id, client, ...)

commons_theme(..., preset = "shiny")
```

Arguments

`id`	The ID of the chat element; must match the `id` of the `shinychat::page_chat()` or `shinychat::chat_ui()` on the page.
`client`	A `commons()` agent. In a deployed app, create the agent inside the server function and pass it to `commons_server()` so each Shiny session gets its own agent state.
`...`	In `commons_server()`, arguments passed to `shinychat::chat_server()`. In `commons_theme()`, named Sass variables forwarded to `shinychat::page_chat_theme()`.
`preset`	A bslib or Bootswatch preset name.

Details

`commons_theme()` bundles the commons chat CSS and JavaScript into an ordinary `bslib::bs_theme()` (via `shinychat::page_chat_theme()`), so it works anywhere a bslib theme does.

Value

`commons_server()` returns the `shinychat::chat_server()` result. `commons_theme()` returns a `bslib::bs_theme()` object.

Examples

```
## Not run:
library(shiny)
library(shinychat)

ui <- page_chat("Assistant", id = "chat", theme = commons_theme())

server <- function(input, output, session) {
  # One agent per session, so each user gets their own agent state
  agent <- commons(
```

```
    ellmer::chat_anthropic(),
    data_sources = data_source(sales = sales)
  )
  commons_server("chat", agent)
}

shinyApp(ui, server)

## End(Not run)
```

`context_layer` *Create a context layer*

Description

A context layer contains text that helps a `commons()` agent interpret its data source.

Usage

```
context_layer(files = character())
```

Arguments

`files`	Character vector of paths to text or Markdown files.

Details

Context is retrieved when relevant. Facts needed in every conversation belong in the `instructions` passed to `commons()`, not here.

Value

A `commons_context_layer` R6 object. Its internals are private and may change without notice.

Examples

```
path <- tempfile(fileext = ".md")
writeLines("Revenue excludes tax unless stated otherwise.", path)
layer <- context_layer(files = path)
```

`data_source` *Create a data source*

Description

A data source is the set of tables available to a `commons()` agent.

Usage

```
data_source(..., tables = NULL, exclude = NULL, dictionary = NULL)
```

Arguments

`...`	A single DBI connection, a single `pins` board, or named data frames to register as tables. When passing data frames, each name becomes a table name the agent can query.
`tables`	Which tables to expose, used when a connection or a board is supplied. For a connection, a character vector of table names, qualified strings like `"schema.table"` or `"catalog.schema.table"`, or `DBI::Id` objects. Defaults to every table returned by `DBI::dbListTables()`. Strings containing dots are interpreted as qualified names, at most three parts; use `DBI::Id(table = "a.b")` for literal table names containing dots. For Snowflake and Databricks connections, a `DBI::Id` ending in `catalog` or `schema` selects every table and view in that namespace. Leaving `tables` unset selects the current schema. A Databricks `hive_metastore` selection must include a schema. Snowflake selections import semantic views, and Databricks selections import metric views, as native trusted metrics and dimensions. Namespace selections read model definitions lazily. Explicitly selected models are read and validated when the data source is created. Databricks wildcard members require concrete column metadata from the warehouse. An exact physical-table selection also imports associated models when every physical dependency is selected. Only public relationships, facts, filters, and instructions are exposed to the agent. For a board, a named character vector of pins to read: the names become table names, and the values are pin names passed to `pins::pin_read()`.
`exclude`	For Snowflake and Databricks namespace selections, optional unqualified object-name globs to omit, such as `"TMP_*"`.
`dictionary`	An optional path to a data dictionary describing the source's tables and columns, in the data-dict.yaml format. See the `Data dictionaries` section.

Details

`data_source()` accepts data in several forms, picked by the class of what you pass:

- A DBI connection is queried as-is. Nothing is copied; the agent queries the database directly.
- Named data frames are loaded into an in-process DuckDB database. Use this when the data isn't already in a database.

- A pins board, e.g. pins::board_connect(), is read into the same in-process database: each pin in tables becomes a table. Pin names are validated against the board at construction (a single listing call), but each pin is downloaded only when its table is first used. Calling the agent's prewarm() method (see commons()) starts a background process that downloads the remaining pins into the local pins cache, so a first use typically only reads an already-downloaded file. Since the pins cache is on disk, prewarm() can also run ahead of deployment to warm the cache the deployed app will read. A table reflects the pin's value at first use and is not refreshed for the lifetime of the data source; if a pin can't be read (e.g. a network failure), the error surfaces at that first use and the read is retried on the next one.

Value

A commons_data_source R6 object. Its internals are private and may change without notice.

Data dictionaries

A data dictionary describes a data source's tables and columns: what each table's rows represent, what its columns mean, allowed values and units, how tables join, and definitions of domain terms. commons uses it to provide business context and governed definitions to the agent. See vignette("commons", package = "commons") for guidance on writing one.

For Snowflake and Databricks sources, a fully qualified dictionary table name matches the same selected relation. A relative name is accepted when it matches only one selected relation. Authored prose takes precedence, while warehouse column types remain authoritative.

A table's entry can also declare definitions: named expressions in the data-dict expression language. commons validates their inferred types and references, compiles them for the source's SQL backend, and makes them available to trusted metric calculations and custom SQL.

Trust

The agent runs only read-only SELECT queries; statements that would modify data or schema (INSERT, UPDATE, DROP, and similar) are rejected before reaching the database. For the in-process DuckDB built from data frames, commons additionally disables extension loading and filesystem access. These are safeguards, not a sandbox: when you supply your own connection, still open it in read-only mode where the backend supports it. Snowflake and Databricks sources snapshot the principal and namespace at creation, and Snowflake its active and secondary roles as well, then reject catalog access and trusted calculations after any of those change. Authored and native semantic material is exposed only after a zero-row query succeeds for the current principal.

Examples

```
src <- data_source(
  sales = data.frame(id = 1:2, revenue = c(100, 200))
)
```

`measure` *Create a measure*

Description

A measure is a trusted calculation inside a `semantic_layer()`. Its function body is ordinary R; its `arguments` schema tells the model what inputs it can supply.

Usage

```
measure(name, description, fn, arguments = list(), title = NULL)
```

Arguments

`name`	Measure name.
`description`	What the measure computes.
`fn`	Function that computes the measure.
`arguments`	A named list of `ellmer::type_string()` and friends describing the arguments the model supplies. Arguments of `fn` not listed here are hidden from the model: they receive a matching data source's connection or keep their defaults. See `semantic_layer()`.
`title`	Human-readable measure title to show in user interfaces. If `NULL`, a title is derived from `name`.

Details

Two return types receive special display handling: ggplots and `gt::gt()` tables are shown directly to the user in the opened measure result.

For custom result content, `fn` can return an ellmer::ContentToolResult. Its `value` is sent to the model and its `extra$display` supplies the shinychat body and card options. Custom HTML is presented inside the standard measure display, after its metadata and arguments. An optional `extra$data` value is made available in the agent's R session and removed from the result before it is returned to ellmer.

Value

A measure object.

See Also

`semantic_layer()` to collect measures into a layer.

Examples

```
table <- data.frame(term = c("Headache", "Nausea"), count = c(7, 5))
table_measure <- measure(
  "adverse_events",
  "Summarize adverse events.",
  function() {
    ellmer::ContentToolResult(
      value = "Headache: 7; Nausea: 5",
      extra = list(
        display = shinychat::tool_result_display(
          html = paste0(
            "<table><tr><td>Headache</td><td>7</td></tr>",
            "<tr><td>Nausea</td><td>5</td></tr></table>"
          )
        ),
        data = table
      )
    )
  }
)
```

`semantic_layer` *Create a semantic layer*

Description

`semantic_layer()` collects trusted calculations for a `commons()` agent. Data dictionary definitions and warehouse semantic models contribute through `data_source()`.

Usage

```
semantic_layer(...)
```

Arguments

`...`	`measure()` objects, lists of measures, or paths to R scripts or directories containing R scripts. Directory searches are not recursive. File and inline measures can be freely mixed.

Value

A `commons_semantic_layer` R6 object. Its internals are private and may change without notice.

Measures from files

Character paths can name R scripts or directories containing them. Functions marked with `@measure` become measures; other functions in those files can be used as helpers.

The roxygen title, description, and `@return` text describe the measure. Each `@param` marks a model-supplied argument and can declare its type: `string`, `integer`, `number`, `boolean`, `enum[value, ...]`, or an array such as `string[]`. Without a declaration, commons infers the type from the default, falling back to `string`.

Measure and helper source is visible in the agent's R session; evaluating a measure's name there prints its definition. Function environments, connections, and credentials are not shared with that session.

Measure arguments

A measure function can take two kinds of arguments:

- Arguments documented with `@param` (or listed in `arguments`, for inline `measure()`s) are supplied by the model.
- Undocumented arguments are supplied by `commons()` when the measure runs. An argument named after a data source receives its connection, even if the argument has a default. Any other undocumented argument keeps its default; if it has no default, `commons()` errors. The model never sees these arguments.

This means a measure can take the connection it needs as an argument rather than relying on a variable defined elsewhere, and you can create a semantic layer before connecting to a database.

For objects that aren't data sources, such as a pins board or an API client, give the argument a default that builds the object, e.g. `board = pins::board_connect()`. Write the default as a call rather than a reference to a variable defined elsewhere, so the measure doesn't depend on where the semantic layer is created.

See Also

`measure()` to define a measure.

Examples

```
semantic_layer(
  measure(
    "order_count",
    "Count of orders.",
    function() 10,
    arguments = list()
  )
)

## Not run:
# In R/semantic_layer.R, `warehouse` has no @param, so commons supplies it:
#
# #' @param region `string` The sales region.
# #' @measure
```

```
# revenue <- function(region, warehouse) {
#   DBI::dbGetQuery(warehouse, ...)
# }

agent <- commons(
  ellmer::chat_anthropic(),
  data_sources = list(warehouse = data_source(DBI::dbConnect(...))),
  semantic_layer = semantic_layer("R/semantic_layer.R")
)

## End(Not run)
```

`trajectory_read` *Read commons trajectories*

Description

`trajectory_read()` reads conversation trajectories captured by `commons()` when `log = TRUE`. Trajectories are recorded as OpenTelemetry spans—see the `log` argument of `commons()` for how capture is enabled—and read back from Posit Connect's content observability store or from local trace files.

This feature requires Connect >= 2026.09.0.

Usage

```
trajectory_read(source = NULL, ..., n = NULL, from = NULL, to = NULL)
```

Arguments

`source`	Where to read trajectories from: • `NULL` (the default) resolves automatically: on Posit Connect, this content's own traces; in a project that has been deployed with rsconnect, the deployed content's traces; otherwise, the local trace directory that `commons()` writes to. • A Connect content GUID, a content URL (`.../content/<guid>/`), a vanity URL (`.../content/<name>/`), or a dashboard URL (`.../connect/#/apps/<guid>/`). • A directory of OTLP NDJSON trace files (`trace-*.jsonl`).
`...`	These dots are for future extensions and must be empty.
`n`	Keep only the `n` most recent conversations, after `from`/`to` filtering. `NULL` (the default) keeps all of them.
`from, to`	Keep only conversations with chat activity at or after `from` and before `to`. Each is a `POSIXct`, a `Date`, or a single string in a standard format like `"2026-07-22"` or `"2026-07-22 14:30:00"`; dates and strings are interpreted in local time. A conversation that continues past `to` is returned with its history as of `to`.

Details

Reading traces from Connect requires the `CONNECT_API_KEY` environment variable (and `CONNECT_SERVER`, when the server can't be inferred from the URL, the project's deployment record, or the sole Connect server registered with rsconnect), and editor-level access to the content: you must own it or be a collaborator. See the `share_with` argument of `commons()`.

Value

A list of conversations, named by conversation id and ordered oldest-first. Each conversation is a list with a `turns` field containing a list of ellmer::Turns and a `last_active` field containing a `POSIXct` giving the time of the conversation's most recent chat activity.

Examples

```
## Not run:
# Read all of the app's local or deployed trajectories, using the
# automatically resolved source.
trajectories <- trajectory_read()

# Read a recent subset.
recent <- trajectory_read(n = 20, from = "2026-07-01")

# Read trajectories for a specific Connect content item or a local trace
# directory.
deployed <- trajectory_read(
  "https://connect.example.com/content/01234567-89ab-cdef-0123-456789abcdef/"
)
local <- trajectory_read("path/to/traces")

## End(Not run)
```

`trajectory_review` *Review commons trajectories*

Description

`trajectory_review()` launches a Shiny app for browsing and annotating conversation trajectories read with `trajectory_read()`. Reviewers can filter questions by date and trust level, inspect complete conversations, flag conversations or individual question-and-answer exchanges, and record notes.

Use the reviewer to assess answer quality, track provenance outcomes over time, record feedback to guide agent improvements, and identify conversations for further review.

Usage

```
trajectory_review(trajectories = trajectory_read(), review_dir = NULL)
```

Arguments

trajectories	A named list of conversations, as returned by trajectory_read().
review_dir	Optional directory where review actions write generated Markdown documents. Defaults to COMMONS_REVIEW_DIR when set and otherwise to commons-reviews in the working directory.

Details

A single reviewer app writes all review documents to review_dir. Pass it directly, set COMMONS_REVIEW_DIR for the current R process with Sys.setenv(), or add it to .Renviron to keep the setting across local R sessions. Without either, reviews land in commons-reviews relative to the app's working directory.

On Posit Connect, opening a new browser session does not reset review files. By default, review documents are written to the app's working directory, where they are replaced on redeployment. Set review_dir to persistent storage when reviews must survive redeployment. Review apps should use one Connect process because separate processes do not coordinate file writes or in-memory review state.

All sessions of one reviewer app share the same flags and notes; review state is not separated by user. Notes record session$user, the login information supplied by the Shiny host, or "unknown" when it is unavailable. Flags do not record who changed them.

Value

A shiny::shinyApp() object. Calling trajectory_review() at the console launches the reviewer; the result can also be served as the last expression of an app.R.

Viewer

The app charts each trust level's share of answers over time. It uses the finest daily, weekly, or monthly grouping that averages at least five answers per displayed bin, or the coarsest available grouping if none does. Weekly and monthly groupings require windows of at least 14 and 60 days, respectively. A question list is grouped by conversation and can be filtered by date and trust level.

Review records

Notes can apply to a whole conversation or to one question-and-answer exchange selected in the transcript. Flags and notes are stored as one generated Markdown document per reviewed conversation and are restored when the viewer reopens.

Each document contains the complete reviewer-visible conversation and its tool activity. YAML frontmatter stores active flags and note history so the reviewer can restore its state and agents can identify flagged conversations, exchanges, and reviewer notes. The Markdown body is the human-readable transcript for joint human-agent review.

Transcript contents

The transcript uses the same commons and shinychat renderer as live conversations, preserving recorded messages and tool activity. Provenance markers are reconstructed from recorded prove-

nance tags, but inline citations are not recreated. Generated review documents list the recorded citation decisions separately.

Trusted-calculation discovery results are omitted because later activity records any selected calculation; other tool results are limited to 50 lines or 20,000 characters.

Trust filters use each answer's provenance tag exactly as `trajectory_read()` recorded it. Missing or conflicting records are omitted rather than inferred.

Logged calls that aren't part of the agent's question-and-answer record are excluded from the viewer. These include shinychat's conversation-title generation and completions with no user turn.

Examples

```
## Not run:
trajectory_review()

trajectory_review(trajectory_read(from = "2026-07-01"))

Sys.setenv(COMMONS_REVIEW_DIR = "/path/to/persistent/reviews")
trajectory_review()

## End(Not run)
```

Index