Package: vitals (via r-universe)

May 24, 2026

Title Large Language Model Evaluation

Version 0.3.0.9000

Description A port of 'Inspect', a widely adopted 'Python' framework for large language model evaluation. Specifically aimed at 'ellmer' users who want to measure the effectiveness of their large language model-based products, the package supports prompt engineering, tool usage, multi-turn dialog, and model graded evaluations.

License MIT + file LICENSE

URL https://github.com/tidyverse/vitals, https://vitals.tidyverse.org

BugReports https://github.com/tidyverse/vitals/issues

Depends R (>= 4.1)

Imports cli, dplyr, ellmer (>= 0.4.0), glue, httpuv, httr2, jsonlite, purrr, R6, rlang, S7, tibble, tidyr, withr

Suggests ggplot2, here, htmltools, knitr, magick, ordinal, rmarkdown, testthat (>= 3.0.0), vcr (>= 2.0.0)

Config/Needs/website tidyverse/tidytemplate, rmarkdown, btw, tidyverse, gt, brms, RcppEigen, broom, gt

Config/testthat/edition 3

Config/testthat/parallel true

Config/usethis/last-upkeep 2025-04-25

Config/testthat/start-first task, translate

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Config/pak/sysreqs make libicu-dev libssl-dev zlib1g-dev

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

Date/Publication 2026-05-15 20:18:34 UTC

RemoteUrl https://github.com/tidyverse/vitals

RemoteRef HEAD

RemoteSha de58d99bc6cffae411b53fcf99f82488b601bf39

Contents

`are` *An R Eval*

Description

An R Eval is a dataset of challenging R coding problems. Each `input` is a question about R code which could be solved on first-read only by experts and, with a chance to read documentation and run some code, by fluent data scientists. Solutions are in `target()` and enable a fluent data scientist to evaluate whether the solution deserves full, partial, or no credit.

Pass this dataset to `Task$new()` to situate it inside of an evaluation task.

Usage

```
are
```

Format

A tibble with 29 rows and 7 columns:

id Character. Unique identifier/title for the code problem.

input Character. The question to be answered.

target Character. The solution, often with a description of notable features of a correct solution.

domain Character. The technical domain (e.g., Data Analysis, Programming, or Authoring).

task Character. Type of task (e.g., Debugging, New feature, or Translation.)

source Character. URL or source of the problem. `NA`s indicate that the problem was written originally for this eval.

knowledge List. Required knowledge/concepts for solving the problem.

Source

Posit Community, GitHub issues, R4DS solutions, etc. For row-level references, see `source`.

Examples

```
are

dplyr::glimpse(are)
```

`generate` *Convert a chat to a solver function*

Description

`generate()` is the simplest possible solver one might use with vitals; it just passes its inputs to the supplied model and returns its raw responses. The inputs are evaluated in parallel, not in the sense of multiple R sessions, but in the sense of multiple, asynchronous HTTP requests using `ellmer::parallel_chat()`. `generate()`'s output can be passed directory to the `solver` argument of Task's `$new()` method.

Usage

```
generate(solver_chat = NULL)
```

Arguments

`solver_chat`	An ellmer chat object, such as from `ellmer::chat_claude()`, or a zero-argument function that returns one.

Value

The output of `generate()` is another function. That function takes in a vector of `inputs`, as well as a solver chat by the name of `solver_chat` with the default supplied to `generate()` itself.

See the documentation for the `solver` argument in Task for more information on the return type.

See Also

`generate_structured()` for structured output extraction.

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)
```

```
  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  # create a new Task
  tsk <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )

  # evaluate the task (runs solver and scorer) and opens
  # the results in the Inspect log viewer (if interactive)
  tsk$eval()

  # $eval() is shorthand for:
  tsk$solve()
  tsk$score()
  tsk$measure()
  tsk$log()
  tsk$view()

  # get the evaluation results as a data frame
  tsk$get_samples()

  # view the task directory with $view() or vitals_view()
  vitals_view()
}

# The `input` column can be a list of 1-row tibbles for per-sample metadata.
# Custom solvers can then extract columns from each input:
shapes_data <- tibble::tibble(
  input = list(
    tibble::tibble(shapes = "square, circle, rhombus", pick = "square"),
    tibble::tibble(shapes = "square, circle, rhombus", pick = "circle")
  ),
  target = c("square", "circle")
)

my_solver <- function(solver_chat = NULL) {
  chat <- solver_chat
  function(inputs, ..., solver_chat = chat) {
    ch <- if (is.function(solver_chat)) solver_chat() else solver_chat$clone()
    prompts <- lapply(inputs, function(inp) {
      paste0("Always pick ", inp$pick, ". Return only that shape.\n\n", inp$shapes)
    })
    res <- ellmer::parallel_chat(ch, prompts, ...)
    list(result = purrr::map_chr(res, \(c) c$last_turn()@text), solver_chat = res)
  }
}
```

generate_structured *Convert a chat to a solver function with structured output*

Description

generate_structured() is a variant of generate() that uses ellmer::parallel_chat_structured() to extract structured data from the model's responses. This allows you to define a schema for the expected output using ellmer's type_*() functions.

Because parallel_chat_structured() returns structured data rather than Chat objects, generate_structured() creates synthetic Chat objects for logging purposes. These "mock" chats contain the input and JSON-serialized output as turns, but won't include actual token usage or timing metadata from the API.

The result field contains JSON-serialized strings for compatibility with existing scorers. The raw structured data is available in $get_samples()$solver_metadata after calling $solve() or $eval().

Usage

```
generate_structured(solver_chat = NULL, type = NULL)
```

Arguments

solver_chat	An ellmer chat object, such as from ellmer::chat_claude(), or a zero-argument function that returns one.
type	A type specification for the extracted data, created with ellmer's type_*() functions (e.g., ellmer::type_object(), ellmer::type_string()). This defines the schema for the structured output.

Value

The output of generate() is another function. That function takes in a vector of inputs, as well as a solver chat by the name of solver_chat with the default supplied to generate() itself.

See the documentation for the solver argument in Task for more information on the return type.

See Also

generate() for unstructured output, ellmer::type_object() and related functions for defining type specifications.

Examples

```
if (FALSE) {
  library(ellmer)

  type_answer <- type_object(
    answer = type_string(
      "The author's first name, with no other formatting."
```

```
  )
)

names <- tibble::tribble(
  ~input,                                  ~target,
  "Name's Josiah, how's it going?",        "Josiah",
  "I'm Lin, what's your name?",            "Lin",
  "My name is Em Fields, how about you?",  "Em"
)

tsk <- Task$new(
  dataset = names,
  solver = generate_structured(
    solver_chat = chat_anthropic(model = "claude-sonnet-4-20250514"),
    type = type_answer
  ),
  scorer = detect_match("any")
)

tsk$eval()

# the result is JSON-serialized for compatibility with scorers
tsk$get_samples()$result

# raw structured data is available in solver_metadata
tsk$get_samples()$solver_metadata

# solver_chat contains synthetic turns for logging
tsk$get_samples()$solver_chat[[1]]
}
```

`scorer_detect` *Scoring with string detection*

Description

The following functions use string pattern detection to score model outputs.

- `detect_includes()`: Determine whether the `target` from the sample appears anywhere inside the model output. Can be case sensitive or insensitive (defaults to the latter).
- `detect_match()`: Determine whether the `target` from the sample appears at the beginning or end of model output (defaults to looking at the end). Has options for ignoring case, whitespace, and punctuation (all are ignored by default).
- `detect_pattern()`: Extract matches of a pattern from the model response and determine whether those matches also appear in `target`.
- `detect_answer()`: Scorer for model output that precedes answers with "ANSWER: ". Can extract letters, words, or the remainder of the line.
- `detect_exact()`: Scorer which will normalize the text of the answer and target(s) and perform an exact matching comparison of the text. This scorer will return `CORRECT` when the answer is an exact match to one or more targets.

Usage

```
detect_includes(case_sensitive = FALSE)

detect_match(
  location = c("end", "begin", "any", "exact"),
  case_sensitive = FALSE
)

detect_pattern(pattern, case_sensitive = FALSE, all = FALSE)

detect_exact(case_sensitive = FALSE)

detect_answer(format = c("line", "word", "letter"))
```

Arguments

case_sensitive	Logical, whether comparisons are case sensitive.
location	Where to look for match: one of "end", "begin", "any", or "exact". Defaults to "end".
pattern	Regular expression pattern to extract answer.
all	Logical: for multiple captures, whether all must match.
format	What to extract after "ANSWER:": "letter", "word", or "line". Defaults to "line".

Value

A function that scores model output based on string matching. Pass the returned value to $eval(scorer). See the documentation for the scorer argument in Task for more information on the return type.

See Also

model_graded_qa() and model_graded_fact() for model-based scoring.

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  # create a new Task
  tsk <- Task$new(
```

```
    dataset = simple_addition,
    solver = generate(solver_chat = chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = detect_includes()
  )

  # evaluate the task (runs solver and scorer)
  tsk$eval()
}
```

---

scorer_model *Model-based scoring*

Description

Model-based scoring makes use of a model to score output from a solver.

- `model_graded_qa()` scores how well a solver answers a question/answer task.
- `model_graded_fact()` determines whether a solver includes a given fact in its response.

The two scorers are quite similar in their implementation, but use a different default `template` to evaluate correctness.

Usage

```
model_graded_qa(
  template = NULL,
  instructions = NULL,
  grade_pattern = "(?i)GRADE\\s*:\\s*([CPI])(.*)$",
  partial_credit = FALSE,
  scorer_chat = NULL
)

model_graded_fact(
  template = NULL,
  instructions = NULL,
  grade_pattern = "(?i)GRADE\\s*:\\s*([CPI])(.*)$",
  partial_credit = FALSE,
  scorer_chat = NULL
)
```

Arguments

`template`	Grading template to use–a `glue()` string which will take substitutions `input`, `answer`, `criterion`, `instructions`.
`instructions`	Grading instructions. If provided, this completely replaces the default instructions, which specify e.g. how the grader should format its output (e.g. `"GRADE: C"`).

grade_pattern	A regex pattern to extract the final grade from the judge model's response.
partial_credit	Whether to allow partial credit.
scorer_chat	An ellmer chat used to grade the model output, e.g. `ellmer::chat_claude()`.

Value

A function that will grade model responses according to the given instructions. See Task's `scorer` argument for a description of the returned function. The functions that `model_graded_qa()` and `model_graded_fact()` output can be passed directly to `$eval()`.

See the documentation for the `scorer` argument in Task for more information on the return type.

See Also

scorer_detect for string detection-based scoring.

Examples

```
# Quality assurance ----------------------------
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  tsk <- Task$new(
    dataset = simple_addition,
    solver = generate(solver_chat = chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )

  tsk$eval()
}

# Factual response -----------------------------
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  r_history <- tibble(
    input = c(
      "Who created the R programming language?",
      "In what year was version 1.0 of R released?"
```

```
    ),
    target = c("Ross Ihaka and Robert Gentleman.", "2000.")
  )

  tsk <- Task$new(
    dataset = r_history,
    solver = generate(solver_chat = chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_fact()
  )

  tsk$eval()
}
```

`Task` *Creating and evaluating tasks*

Description

Evaluation `Task`s provide a flexible data structure for evaluating LLM-based tools.

1. **Datasets** contain a set of labelled samples. Datasets are just a tibble with columns `input` and `target`, where `input` is a prompt and `target` is either literal value(s) or grading guidance.
2. **Solvers** evaluate the `input` in the dataset and produce a final result.
3. **Scorers** evaluate the final output of solvers. They may use text comparisons (like `detect_match()`), model grading (like `model_graded_qa()`), or other custom schemes.

The usual flow of LLM evaluation with Tasks calls `$new()` and then `$eval()`. `$eval()` just calls `$solve()`, `$score()`, `$measure()`, `$log()`, and `$view()` in order. The remaining methods are generally only recommended for expert use.

Public fields

`dir` The directory where evaluation logs will be written to. Defaults to `vitals_log_dir()`.

`metrics` A named vector of metric values resulting from `$measure()` (called inside of `$eval()`). Will be `NULL` if metrics have yet to be applied.

Methods

Public methods:

- `Task$new()`
- `Task$eval()`
- `Task$get_samples()`
- `Task$solve()`
- `Task$score()`
- `Task$measure()`
- `Task$log()`

- Task$view()
- Task$set_solver()
- Task$set_scorer()
- Task$set_metrics()
- Task$get_cost()
- Task$clone()

Method `new()`**:** The typical flow of LLM evaluation with vitals tends to involve first calling this method and then `$eval()` on the resulting object.

Usage:

```
Task$new(
  dataset,
  solver,
  scorer,
  metrics = NULL,
  epochs = NULL,
  name = deparse(substitute(dataset)),
  dir = vitals_log_dir()
)
```

Arguments:

`dataset` A tibble with, minimally, columns `input` and `target`. The `input` column can be either a character vector or a list-column of 1-row tibbles. Using 1-row tibbles allows per-sample customization by including additional metadata that custom solvers can access.

`solver` A function that takes a vector of inputs from the dataset's `input` column as its first argument and determines values approximating `dataset$target`. Its return value must be a list with the following elements:

- `result` - A character vector of the final responses, with the same length as `dataset$input`.
- `solver_chat` - A list of ellmer Chat objects that were used to solve each input, also with the same length as `dataset$input`.

Additional output elements can be included in a slot `solver_metadata` that has the same length as `dataset$input`, which will be logged in `solver_metadata`.
Additional arguments can be passed to the solver via `$solve(...)` or `$eval(...)`. See the definition of `generate()` for a function that outputs a valid solver that just passes inputs to ellmer Chat objects' `$chat()` method in parallel.

`scorer` A function that evaluates how well the solver's return value approximates the corresponding elements of `dataset$target`. The function should take in the `$get_samples()` slot of a Task object and return a list with the following elements:

- `score` - A vector of scores with length equal to `nrow(samples)`. Built-in scorers return ordered factors with levels `I` < `P` (optionally) < `C` (standing for "Incorrect", "Partially Correct", and "Correct"). If your scorer returns this output type, the package will automatically calculate metrics.

Optionally:

- `scorer_chat` - If your scorer makes use of ellmer, also include a list of ellmer Chat objects that were used to score each result, also with length `nrow(samples)`.
- `scorer_metadata` - Any intermediate results or other values that you'd like to be stored in the persistent log. This should also have length equal to `nrow(samples)`.

Scorers will probably make use of `samples$input`, `samples$target`, and `samples$result` specifically. See model-based scoring for examples.

`metrics` A named list of functions that take in a vector of scores (as in `task$get_samples()$score`) and output a single numeric value.

`epochs` The number of times to repeat each sample. Evaluate each sample multiple times to better quantify variation. Optional, defaults to `1L`. The value of `epochs` supplied to `$eval()` or `$score()` will take precedence over the value in `$new()`.

`name` A name for the evaluation task. Defaults to `deparse(substitute(dataset))`.

`dir` Directory where logs should be stored.

Returns: A new Task object.

Method `eval()`**:** Evaluates the task by running the solver, scorer, logging results, and viewing (if interactive). This method works by calling `$solve()`, `$score()`, `$log()`, and `$view()` in sequence.

The typical flow of LLM evaluation with vitals tends to involve first calling `$new()` and then this method on the resulting object.

Usage:

```
Task$eval(..., epochs = NULL, view = interactive())
```

Arguments:

`...` Additional arguments passed to the solver and scorer functions. All arguments must be named. Arguments are routed based on function signatures: if an argument name matches a parameter in the solver, it goes to the solver; if it matches a parameter in the scorer, it goes to the scorer. Arguments matching both go to both. Unmatched arguments are passed to any function with `...` in its signature. An error is raised if an argument matches neither function and neither accepts `...`.

`epochs` The number of times to repeat each sample. Evaluate each sample multiple times to better quantify variation. Optional, defaults to `1L`. The value of `epochs` supplied to `$eval()` or `$score()` will take precedence over the value in `$new()`.

`view` Automatically open the viewer after evaluation (defaults to TRUE if interactive, FALSE otherwise).

Returns: The Task object (invisibly)

Method `get_samples()`**:** The task's samples represent the evaluation in a data frame format. `vitals_bind()` row-binds the output of this function called across several tasks.

Usage:

```
Task$get_samples()
```

Returns: A tibble representing the evaluation. Based on the `dataset`, `epochs` may duplicate rows, and the solver and scorer will append columns to this data.

Method `solve()`**:** Solve the task by running the solver

Usage:

```
Task$solve(..., epochs = NULL)
```

Arguments:

`...` Additional arguments passed to the solver function.

epochs The number of times to repeat each sample. Evaluate each sample multiple times to better quantify variation. Optional, defaults to 1L. The value of epochs supplied to $eval() or $score() will take precedence over the value in $new().

Returns: The Task object (invisibly)

Method score(): Score the task by running the scorer and then applying metrics to its results.

Usage:

```
Task$score(...)
```

Arguments:

... Additional arguments passed to the scorer function.

Returns: The Task object (invisibly)

Method measure(): Applies metrics to a scored Task.

Usage:

```
Task$measure()
```

Returns: The Task object (invisibly)

Method log(): Log the task to a directory.

Note that, if an VITALS_LOG_DIR envvar is set, this will happen automatically in $eval().

Usage:

```
Task$log(dir = self$dir)
```

Arguments:

dir The directory to write the log to.

Returns: The path to the logged file, invisibly.

Method view(): View the task results in the Inspect log viewer

Usage:

```
Task$view()
```

Returns: The Task object (invisibly)

Method set_solver(): Set the solver function

Usage:

```
Task$set_solver(solver)
```

Arguments:

solver A function that takes a vector of inputs from the dataset's input column as its first argument and determines values approximating dataset$target. Its return value must be a list with the following elements:

- result - A character vector of the final responses, with the same length as dataset$input.
- solver_chat - A list of ellmer Chat objects that were used to solve each input, also with the same length as dataset$input.

Additional output elements can be included in a slot solver_metadata that has the same length as dataset$input, which will be logged in solver_metadata.
Additional arguments can be passed to the solver via $solve(...) or $eval(...). See the definition of generate() for a function that outputs a valid solver that just passes inputs to ellmer Chat objects' $chat() method in parallel.

Returns: The Task object (invisibly)

Method `set_scorer()`**:** Set the scorer function

Usage:

```
Task$set_scorer(scorer)
```

Arguments:

`scorer` A function that evaluates how well the solver's return value approximates the corresponding elements of `dataset$target`. The function should take in the `$get_samples()` slot of a Task object and return a list with the following elements:

- `score` - A vector of scores with length equal to `nrow(samples)`. Built-in scorers return ordered factors with levels `I` < `P` (optionally) < `C` (standing for "Incorrect", "Partially Correct", and "Correct"). If your scorer returns this output type, the package will automatically calculate metrics.

Optionally:

- `scorer_chat` - If your scorer makes use of ellmer, also include a list of ellmer Chat objects that were used to score each result, also with length `nrow(samples)`.
- `scorer_metadata` - Any intermediate results or other values that you'd like to be stored in the persistent log. This should also have length equal to `nrow(samples)`.

Scorers will probably make use of `samples$input`, `samples$target`, and `samples$result` specifically. See model-based scoring for examples.

Returns: The Task object (invisibly)

Method `set_metrics()`**:** Set the metrics that will be applied in `$measure()` (and thus `$eval()`).

Usage:

```
Task$set_metrics(metrics)
```

Arguments:

`metrics` A named list of functions that take in a vector of scores (as in `task$get_samples()$score`) and output a single numeric value.

Returns: The Task (invisibly)

Method `get_cost()`**:** The cost of this eval This is a wrapper around ellmer's `$token_usage()` function. That function is called at the beginning and end of each call to `$solve()` and `$score()`; this function returns the cost inferred by taking the differences in values of `$token_usage()` over time.

Usage:

```
Task$get_cost()
```

Returns: A tibble displaying the cost of solving and scoring the evaluation by model, separately for the solver and scorer.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Task$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

generate() for the simplest possible solver, and scorer_model and scorer_detect for two built-in approaches to scoring.

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  # create a new Task
  tsk <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )

  # evaluate the task (runs solver and scorer) and opens
  # the results in the Inspect log viewer (if interactive)
  tsk$eval()

  # $eval() is shorthand for:
  tsk$solve()
  tsk$score()
  tsk$measure()
  tsk$log()
  tsk$view()

  # get the evaluation results as a data frame
  tsk$get_samples()

  # view the task directory with $view() or vitals_view()
  vitals_view()
}

# The `input` column can be a list of 1-row tibbles for per-sample metadata.
# Custom solvers can then extract columns from each input:
shapes_data <- tibble::tibble(
  input = list(
    tibble::tibble(shapes = "square, circle, rhombus", pick = "square"),
    tibble::tibble(shapes = "square, circle, rhombus", pick = "circle")
  ),
  target = c("square", "circle")
)
```

```
my_solver <- function(solver_chat = NULL) {
  chat <- solver_chat
  function(inputs, ..., solver_chat = chat) {
    ch <- if (is.function(solver_chat)) solver_chat() else solver_chat$clone()
    prompts <- lapply(inputs, function(inp) {
      paste0("Always pick ", inp$pick, ". Return only that shape.\n\n", inp$shapes)
    })
    res <- ellmer::parallel_chat(ch, prompts, ...)
    list(result = purrr::map_chr(res, \(c) c$last_turn()@text), solver_chat = res)
  }
}
```

`vitals_bind` *Concatenate task samples for analysis*

Description

Combine multiple Task objects into a single tibble for comparison.

This function takes multiple (optionally named) Task objects and row-binds their `$get_samples()` together, adding a `task` column to identify the source of each row. The resulting tibble nests additional columns into a `metadata` column and is ready for further analysis.

Usage

```
vitals_bind(...)
```

Arguments

`...`	Task objects to combine, optionally named.

Value

A tibble with the combined samples from all tasks, with a `task` column indicating the source and a nested `metadata` column containing additional fields.

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
```

```
  )

  tsk1 <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )
  tsk1$eval()

  tsk2 <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = detect_includes()
  )
  tsk2$eval()

  combined <- vitals_bind(model_graded = tsk1, string_detection = tsk2)
}
```

`vitals_bundle` *Prepare logs for deployment*

Description

This function creates a standalone bundle of the Inspect viewer with log files that can be deployed statically. It copies the UI viewer files, log files, and generates the necessary configuration files.

Usage

```
vitals_bundle(log_dir = vitals_log_dir(), output_dir = NULL, overwrite = FALSE)
```

Arguments

`log_dir`	Path to the directory containing log files. Defaults to `vitals_log_dir()`.
`output_dir`	Path to the directory where the bundled output will be placed.
`overwrite`	Whether to overwrite an existing output directory. Defaults to FALSE.

Value

Invisibly returns the output directory path. That directory contains:

```
output_dir
|-- index.html
|-- robots.txt
|-- assets
    |--  ..
|-- logs
    |--  ..
```

`robots.txt` prevents crawlers from indexing the viewer. That said, many crawlers only read the `robots.txt` at the root directory of a package, so the file will likely be ignored if this folder isn't the root directory of the deployed page. `assets/` is the bundled source for the viewer. `logs/` is the `log_dir` as well as a `listing.json`, which is a manifest file for the directory.

Deployment

This function generates a directory that's ready for deployment to any static web server such as GitHub Pages, S3 buckets, or Netlify. If you have a connection to Posit Connect configured, you can deploy a directory of log files with the following:

```
tmp_dir <- withr::local_tempdir()
vitals_bundle(output_dir = tmp_dir, overwrite = TRUE)
rsconnect::deployApp(tmp_dir)
```

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  tsk <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )

  tsk$eval()

  output_dir <- tempdir()
  vitals_bundle(output_dir = output_dir, overwrite = TRUE)
}
```

---

vitals_log_dir *The log directory*

Description

vitals supports the `VITALS_LOG_DIR` environment variable, which sets a default directory to write logs to in Task's `$eval()` and `$log()` methods.

Usage

```
vitals_log_dir()

vitals_log_dir_set(dir)
```

Arguments

dir	A directory to configure the environment variable VITALS_LOG_DIR to.

Value

Both vitals_log_dir() and vitals_log_dir_set() return the current value of the environment variable VITALS_LOG_DIR. vitals_log_dir_set() additionally sets it to a new value.

To set this variable in every new R session, you might consider adding it to your .Rprofile, perhaps with usethis::edit_r_profile().

Examples

```
vitals_log_dir()

dir <- tempdir()

vitals_log_dir_set(dir)

vitals_log_dir()
```

---

vitals_view *Interactively view local evaluation logs*

Description

vitals bundles the Inspect log viewer, an interactive app for exploring evaluation logs. Supply a path to a directory of tasks written to json. For individual Task objects, use the $view() method instead.

Usage

```
vitals_view(dir = vitals_log_dir(), host = "127.0.0.1", port = NULL)
```

Arguments

dir	Path to a directory containing task eval logs.
host	Host to serve on. Defaults to "127.0.0.1".
port	Port to serve on. If NULL, will find a random available port.

Value

The server object (invisibly)

Examples

```
if (!identical(Sys.getenv("ANTHROPIC_API_KEY"), "")) {
  # set the log directory to a temporary directory
  withr::local_envvar(VITALS_LOG_DIR = withr::local_tempdir())

  library(ellmer)
  library(tibble)

  simple_addition <- tibble(
    input = c("What's 2+2?", "What's 2+3?"),
    target = c("4", "5")
  )

  # create a new Task
  tsk <- Task$new(
    dataset = simple_addition,
    solver = generate(chat_claude(model = "claude-sonnet-4-5-20250929")),
    scorer = model_graded_qa()
  )

  # evaluate the task (runs solver and scorer) and opens
  # the results in the Inspect log viewer (if interactive)
  tsk$eval()

  # $eval() is shorthand for:
  tsk$solve()
  tsk$score()
  tsk$measure()
  tsk$log()
  tsk$view()

  # get the evaluation results as a data frame
  tsk$get_samples()

  # view the task directory with $view() or vitals_view()
  vitals_view()
}

# The `input` column can be a list of 1-row tibbles for per-sample metadata.
# Custom solvers can then extract columns from each input:
shapes_data <- tibble::tibble(
  input = list(
    tibble::tibble(shapes = "square, circle, rhombus", pick = "square"),
    tibble::tibble(shapes = "square, circle, rhombus", pick = "circle")
  ),
  target = c("square", "circle")
)

my_solver <- function(solver_chat = NULL) {
  chat <- solver_chat
  function(inputs, ..., solver_chat = chat) {
    ch <- if (is.function(solver_chat)) solver_chat() else solver_chat$clone()
```

```
    prompts <- lapply(inputs, function(inp) {
      paste0("Always pick ", inp$pick, ". Return only that shape.\n\n", inp$shapes)
    })
    res <- ellmer::parallel_chat(ch, prompts, ...)
    list(result = purrr::map_chr(res, \(c) c$last_turn()@text), solver_chat = res)
  }
}
```

Index