Package ‘ellmer’

May 18, 2025

Title Chat with Large Language Models

Version 0.2.0

Description Chat with large language models from a range of providers
including 'Claude' <https://claude.ai>, 'OpenAI'
<https://chatgpt.com>, and more. Supports streaming, asynchronous
calls, tool calling, and structured data extraction.

License MIT + file LICENSE

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

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

Depends R (>= 4.1)

Imports cli, coro (>= 1.1.0), glue, httr2 (>= 1.1.1), jsonlite, later
(>= 1.4.0), lifecycle, promises (>= 1.3.1), R6, rlang (>=
1.1.0), S7 (>= 0.2.0)

Suggests base64enc, connectcreds, curl (>= 6.0.1), gargle, gitcreds,
knitr, magick, openssl, paws.common, rmarkdown, shiny,
shinychat (>= 0.2.0), testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate, rmarkdown

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first test-provider-*

Encoding UTF-8

RoxygenNote 7.3.2

Collate 'utils-S7.R' 'types.R' 'tools-def.R' 'content.R' 'provider.R'
'as-json.R' 'batch-chat.R' 'chat-parallel.R'
'chat-structured.R' 'utils-coro.R' 'chat.R' 'content-image.R'
'content-pdf.R' 'turns.R' 'content-tools.R' 'deprecated.R'
'ellmer-package.R' 'httr2.R' 'import-standalone-obj-type.R'
'import-standalone-purrr.R' 'import-standalone-types-check.R'
'interpolate.R' 'params.R' 'provider-openai.R'

'provider-azure.R' 'provider-bedrock.R' 'provider-claude.R'
'provider-gemini.R' 'provider-cloudflare.R' 'provider-cortex.R'
'provider-databricks.R' 'provider-deepseek.R'
'provider-gemini-upload.R' 'provider-github.R'
'provider-groq.R' 'provider-huggingface.r' 'provider-mistral.R'
'provider-ollama.R' 'provider-openrouter.R'
'provider-perplexity.R' 'provider-portkey.R'
'provider-snowflake.R' 'provider-vllm.R' 'shiny.R' 'tokens.R'
'tools-def-auto.R' 'utils-callbacks.R' 'utils-cat.R'
'utils-merge.R' 'utils-prettytime.R' 'utils.R' 'zzz.R'

NeedsCompilation no

Author Hadley Wickham [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4757-117X>),
Joe Cheng [aut],
Aaron Jacobs [aut],
Garrick Aden-Buie [aut] (ORCID:
<https://orcid.org/0000-0002-7111-0077>),
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2025-05-17 22:20:02 UTC

Contents

`batch_chat` *Submit multiple chats in one batch*

Description

[Experimental]

`batch_chat()` and `batch_chat_structured()` currently only work with `chat_openai()` and `chat_anthropic()`. They use the OpenAI and Anthropic batch APIs which allow you to submit multiple requests simultaneously. The results can take up to 24 hours to complete, but in return you pay 50% less than usual (but note that ellmer doesn't include this discount in its pricing metadata). If you want to get results back more quickly, or you're working with a different provider, you may want to use `parallel_chat()` instead.

Since batched requests can take a long time to complete, `batch_chat()` requires a file path that is used to store information about the batch so you never lose any work. You can either set `wait = FALSE` or simply interrupt the waiting process, then later, either call `batch_chat()` to resume where you left off or call `batch_chat_completed()` to see if the results are ready to retrieve. `batch_chat()` will store the chat responses in this file, so you can either keep it around to cache the results, or delete it to free up disk space.

This API is marked as experimental since I don't yet know how to handle errors in the most helpful way. Fortunately they don't seem to be common, but if you have ideas, please let me know!

Usage

```
batch_chat(chat, prompts, path, wait = TRUE)

batch_chat_structured(
  chat,
  prompts,
  path,
  type,
  wait = TRUE,
  convert = TRUE,
  include_tokens = FALSE,
  include_cost = FALSE
)

batch_chat_completed(chat, prompts, path)
```

Arguments

`chat` A base chat object.

`prompts` A vector created by `interpolate()` or a list of character vectors.

`path` Path to file (with `.json` extension) to store state.

The file records a hash of the provider, the prompts, and the existing chat turns. If you attempt to reuse the same file with any of these being different, you'll get an error.

`wait` If `TRUE`, will wait for batch to complete. If `FALSE`, it will return `NULL` if the batch is not complete, and you can retrieve the results later by re-running `batch_chat()` when `batch_chat_completed()` is `TRUE`.

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`convert` If `TRUE`, automatically convert from JSON lists to R data types using the schema. This typically works best when `type` is `type_object()` as this will give you a data frame with one column for each property. If `FALSE`, returns a list.

`include_tokens` If `TRUE`, and the result is a data frame, will add `input_tokens` and `output_tokens` columns giving the total input and output tokens for each prompt.

`include_cost` If `TRUE`, and the result is a data frame, will add `cost` column giving the cost of each prompt.

Examples

```
chat <- chat_openai(model = "gpt-4.1-nano")

# Chat ----------------------------------------------------------------------

prompts <- interpolate("What do people from {{state.name}} bring to a potluck dinner?")
## Not run:
chats <- batch_chat(chat, prompts, path = "potluck.json")
chats

## End(Not run)

# Structured data -----------------------------------------------------------
prompts <- list(
  "I go by Alex. 42 years on this planet and counting.",
  "Pleased to meet you! I'm Jamal, age 27.",
  "They call me Li Wei. Nineteen years young.",
  "Fatima here. Just celebrated my 35th birthday last week.",
  "The name's Robert - 51 years old and proud of it.",
  "Kwame here - just hit the big 5-0 this year."
)
type_person <- type_object(name = type_string(), age = type_number())
## Not run:
data <- batch_chat_structured(
  chat = chat,
  prompts = prompts,
  path = "people-data.json",
  type = type_person
)
data

## End(Not run)
```

Chat   *A chat*

Description

A Chat is a sequence of user and assistant Turns sent to a specific Provider. A Chat is a mutable R6 object that takes care of managing the state associated with the chat; i.e. it records the messages that you send to the server, and the messages that you receive back. If you register a tool (i.e. an R function that the assistant can call on your behalf), it also takes care of the tool loop.

You should generally not create this object yourself, but instead call chat_openai() or friends instead.

Value

A Chat object

Methods

Public methods:

- Chat$new()
- Chat$get_turns()
- Chat$set_turns()
- Chat$add_turn()
- Chat$get_system_prompt()
- Chat$get_model()
- Chat$set_system_prompt()
- Chat$get_tokens()
- Chat$get_cost()

- `Chat$last_turn()`
- `Chat$chat()`
- `Chat$chat_structured()`
- `Chat$chat_structured_async()`
- `Chat$chat_async()`
- `Chat$stream()`
- `Chat$stream_async()`
- `Chat$register_tool()`
- `Chat$get_provider()`
- `Chat$get_tools()`
- `Chat$set_tools()`
- `Chat$on_tool_request()`
- `Chat$on_tool_result()`
- `Chat$extract_data()`
- `Chat$extract_data_async()`
- `Chat$clone()`

Method `new()`**:**

Usage:

```
Chat$new(provider, system_prompt = NULL, echo = "none")
```

Arguments:

`provider` A provider object.

`system_prompt` System prompt to start the conversation with.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Method `get_turns()`**:** Retrieve the turns that have been sent and received so far (optionally starting with the system prompt, if any).

Usage:

```
Chat$get_turns(include_system_prompt = FALSE)
```

Arguments:

`include_system_prompt` Whether to include the system prompt in the turns (if any exists).

Method `set_turns()`**:** Replace existing turns with a new list.

Usage:

```
Chat$set_turns(value)
```

Arguments:

`value` A list of Turns.

Method `add_turn()`**:** Add a pair of turns to the chat.

Usage:

```
Chat$add_turn(user, system)
```

Arguments:

`user` The user Turn.

`system` The system Turn.

Method `get_system_prompt():` If set, the system prompt, it not, `NULL`.

Usage:

```
Chat$get_system_prompt()
```

Method `get_model():` Retrieve the model name

Usage:

```
Chat$get_model()
```

Method `set_system_prompt():` Update the system prompt

Usage:

```
Chat$set_system_prompt(value)
```

Arguments:

`value` A character vector giving the new system prompt

Method `get_tokens():` A data frame with a `tokens` column that proides the number of input tokens used by user turns and the number of output tokens used by assistant turns.

Usage:

```
Chat$get_tokens(include_system_prompt = FALSE)
```

Arguments:

`include_system_prompt` Whether to include the system prompt in the turns (if any exists).

Method `get_cost():` The cost of this chat

Usage:

```
Chat$get_cost(include = c("all", "last"))
```

Arguments:

`include` The default, `"all"`, gives the total cumulative cost of this chat. Alternatively, use `"last"` to get the cost of just the most recent turn.

Method `last_turn():` The last turn returned by the assistant.

Usage:

```
Chat$last_turn(role = c("assistant", "user", "system"))
```

Arguments:

`role` Optionally, specify a role to find the last turn with for the role.

Returns: Either a `Turn` or `NULL`, if no turns with the specified role have occurred.

Method `chat():` Submit input to the chatbot, and return the response as a simple string (probably Markdown).

Usage:

```
Chat$chat(..., echo = NULL)
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images (see `content_image_file()` and `content_image_url()`.

`echo` Whether to emit the response to stdout as it is received. If `NULL`, then the value of `echo` set when the chat object was created will be used.

Method `chat_structured()`**:** Extract structured data

Usage:

```
Chat$chat_structured(..., type, echo = "none", convert = TRUE)
```

Arguments:

`...` The input to send to the chatbot. Will typically include the phrase "extract structured data".

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`echo` Whether to emit the response to stdout as it is received. Set to "text" to stream JSON data as it's generated (not supported by all providers).

`convert` Automatically convert from JSON lists to R data types using the schema. For example, this will turn arrays of objects into data frames and arrays of strings into a character vector.

Method `chat_structured_async()`**:** Extract structured data, asynchronously. Returns a promise that resolves to an object matching the type specification.

Usage:

```
Chat$chat_structured_async(..., type, echo = "none")
```

Arguments:

`...` The input to send to the chatbot. Will typically include the phrase "extract structured data".

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`echo` Whether to emit the response to stdout as it is received. Set to "text" to stream JSON data as it's generated (not supported by all providers).

Method `chat_async()`**:** Submit input to the chatbot, and receive a promise that resolves with the response all at once. Returns a promise that resolves to a string (probably Markdown).

Usage:

```
Chat$chat_async(..., tool_mode = c("concurrent", "sequential"))
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`tool_mode` Whether tools should be invoked one-at-a-time (`"sequential"`) or concurrently (`"concurrent"`). Sequential mode is best for interactive applications, especially when a tool may involve an interactive user interface. Concurrent mode is the default and is best suited for automated scripts or non-interactive applications.

Method `stream()`**:** Submit input to the chatbot, returning streaming results. Returns A coro generator that yields strings. While iterating, the generator will block while waiting for more content from the chatbot.

Usage:

```
Chat$stream(..., stream = c("text", "content"))
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`stream` Whether the stream should yield only `"text"` or ellmer's rich content types. When `stream = "content"`, `stream()` yields Content objects.

Method `stream_async()`**:** Submit input to the chatbot, returning asynchronously streaming results. Returns a coro async generator that yields string promises.

Usage:

```
Chat$stream_async(
  ...,
  tool_mode = c("concurrent", "sequential"),
  stream = c("text", "content")
)
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`tool_mode` Whether tools should be invoked one-at-a-time (`"sequential"`) or concurrently (`"concurrent"`). Sequential mode is best for interactive applications, especially when a tool may involve an interactive user interface. Concurrent mode is the default and is best suited for automated scripts or non-interactive applications.

`stream` Whether the stream should yield only `"text"` or ellmer's rich content types. When `stream = "content"`, `stream()` yields Content objects.

Method `register_tool()`**:** Register a tool (an R function) that the chatbot can use. If the chatbot decides to use the function, ellmer will automatically call it and submit the results back.

The return value of the function. Generally, this should either be a string, or a JSON-serializable value. If you must have more direct control of the structure of the JSON that's returned, you can return a JSON-serializable value wrapped in `base::I()`, which ellmer will leave alone until the entire request is JSON-serialized.

Usage:

```
Chat$register_tool(tool_def)
```

Arguments:

`tool_def` Tool definition created by `tool()`.

Method `get_provider()`**:** Get the underlying provider object. For expert use only.

Usage:

```
Chat$get_provider()
```

Method `get_tools()`**:** Retrieve the list of registered tools.

Usage:

```
Chat$get_tools()
```

Method `set_tools()`**:** Sets the available tools. For expert use only; most users should use `register_tool()`.

Usage:

```
Chat$set_tools(tools)
```

Arguments:

`tools` A list of tool definitions created with `tool()`.

Method `on_tool_request()`**:** Register a callback for a tool request event.

Usage:

```
Chat$on_tool_request(callback)
```

Arguments:

`callback` A function to be called when a tool request event occurs, which must have `request` as its only argument.

Returns: A function that can be called to remove the callback.

Method `on_tool_result()`**:** Register a callback for a tool result event.

Usage:

```
Chat$on_tool_result(callback)
```

Arguments:

`callback` A function to be called when a tool result event occurs, which must have `result` as its only argument.

Returns: A function that can be called to remove the callback.

Method `extract_data()`**:** **[Deprecated]** Deprecated in favour of `$chat_structured()`.

Usage:

```
Chat$extract_data(...)
```

Arguments:

`...` See `$chat_structured()`

Method `extract_data_async()`**:** **[Deprecated]**

Usage:

```
Chat$extract_data_async(...)
```

Arguments:

`...` See `$chat_structured_async()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
chat <- chat_openai(echo = TRUE)
chat$chat("Tell me a funny joke")
```

`chat_anthropic` *Chat with an Anthropic Claude model*

Description

Anthropic provides a number of chat based models under the Claude moniker. Note that a Claude Pro membership does not give you the ability to call models via the API; instead, you will need to sign up (and pay for) a developer account.

Usage

```
chat_anthropic(
  system_prompt = NULL,
  params = NULL,
  max_tokens = deprecated(),
  model = NULL,
  api_args = list(),
  base_url = "https://api.anthropic.com/v1",
  beta_headers = character(),
  api_key = anthropic_key(),
  echo = NULL
)

models_anthropic(
  base_url = "https://api.anthropic.com/v1",
  api_key = anthropic_key()
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`params` Common model parameters, usually created by `params()`.

`max_tokens` Maximum number of tokens to generate before stopping.

`model` The model to use for the chat (defaults to "claude-3-7-sonnet-latest"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_anthropic()` to see all options.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`beta_headers` Optionally, a character vector of beta headers to opt-in claude features that are still in beta.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `ANTHROPIC_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
chat <- chat_anthropic()
chat$chat("Tell me three jokes about statisticians")
```

`chat_aws_bedrock` *Chat with an AWS bedrock model*

Description

AWS Bedrock provides a number of language models, including those from Anthropic's Claude, using the Bedrock Converse API.

Authentication:

Authentication is handled through {paws.common}, so if authentication does not work for you automatically, you'll need to follow the advice at `https://www.paws-r-sdk.com/#credentials`. In particular, if your org uses AWS SSO, you'll need to run `aws sso login` at the terminal.

Usage

```
chat_aws_bedrock(
  system_prompt = NULL,
  model = NULL,
  profile = NULL,
  api_args = list(),
  echo = NULL
)

models_aws_bedrock(profile = NULL)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`model` The model to use for the chat (defaults to "anthropic.claude-3-5-sonnet-20240620-v1:0"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_models_aws_bedrock()` to see all options. .

While ellmer provides a default model, there's no guarantee that you'll have access to it, so you'll need to specify a model that you can. If you're using cross-region inference, you'll need to use the inference profile ID, e.g. `model="us.anthropic.claude-3-5-so`

`profile` AWS profile to use.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Some useful arguments include:

```
api_args = list(
  inferenceConfig = list(
    maxTokens = 100,
    temperature = 0.7,
    topP = 0.9,
    topK = 20
  )
)
```

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
# Basic usage
chat <- chat_aws_bedrock()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_azure_openai` *Chat with a model hosted on Azure OpenAI*

Description

The Azure OpenAI server hosts a number of open source models as well as proprietary models from OpenAI.

Authentication:

`chat_azure_openai()` supports API keys and the `credentials` parameter, but it also makes use of:

- Azure service principals (when the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables are set).
- Interactive Entra ID authentication, like the Azure CLI.
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Usage

```
chat_azure_openai(
  endpoint = azure_endpoint(),
  deployment_id,
  params = NULL,
  api_version = NULL,
  system_prompt = NULL,
  api_key = NULL,
  token = deprecated(),
  credentials = NULL,
  api_args = list(),
  echo = c("none", "output", "all")
)
```

Arguments

`endpoint` Azure OpenAI endpoint url with protocol and hostname, i.e. `https://{your-resource-name}.openai.` Defaults to using the value of the `AZURE_OPENAI_ENDPOINT` envinronment variable.

`deployment_id` Deployment id for the model you want to use.

`params` Common model parameters, usually created by `params()`.

`api_version` The API version to use.

`system_prompt` A system prompt to set the behavior of the assistant.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `AZURE_OPENAI_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`token` **[Deprecated]** A literal Azure token to use for authentication. Deprecated in favour of ambient Azure credentials or an explicit `credentials` argument.

`credentials` A list of authentication headers to pass into `httr2::req_headers()`, a function that returns them, or `NULL` to use `token` or `api_key` to generate these headers instead. This is an escape hatch that allows users to incorporate Azure credentials generated by other packages into **ellmer**, or to manage the lifetime of credentials that need to be refreshed.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_azure_openai(deployment_id = "gpt-4o-mini")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_cloudflare` *Chat with a model hosted on CloudFlare*

Description

Cloudflare works AI hosts a variety of open-source AI models. To use the Cloudflare API, you must have an Account ID and an Access Token, which you can obtain by following these instructions.

Known limitations:

- Tool calling does not appear to work.
- Images don't appear to work.

Usage

```
chat_cloudflare(
  account = cloudflare_account(),
  system_prompt = NULL,
  params = NULL,
  api_key = cloudflare_key(),
  model = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`account` The Cloudflare account ID. Taken from the `CLOUDFLARE_ACCOUNT_ID` env var, if defined.

`system_prompt` A system prompt to set the behavior of the assistant.

`params` Common model parameters, usually created by `params()`.

`api_key` The API key to use for authentication. You generally should not supply this directly, but instead set the `HUGGINGFACE_API_KEY` environment variable.

`model` The model to use for the chat (defaults to "meta-llama/Llama-3.3-70b-instruct-fp8-fast"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_cloudflare()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_cortex_analyst *Create a chatbot that speaks to the Snowflake Cortex Analyst*

Description

Chat with the LLM-powered Snowflake Cortex Analyst.

Authentication:

`chat_cortex_analyst()` picks up the following ambient Snowflake credentials:

- A static OAuth token defined via the `SNOWFLAKE_TOKEN` environment variable.
- Key-pair authentication credentials defined via the `SNOWFLAKE_USER` and `SNOWFLAKE_PRIVATE_KEY` (which can be a PEM-encoded private key or a path to one) environment variables.
- Posit Workbench-managed Snowflake credentials for the corresponding `account`.
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Known limitations:

Unlike most comparable model APIs, Cortex does not take a system prompt. Instead, the caller must provide a "semantic model" describing available tables, their meaning, and verified queries that can be run against them as a starting point. The semantic model can be passed as a YAML string or via reference to an existing file in a Snowflake Stage.

Note that Cortex does not support multi-turn, so it will not remember previous messages. Nor does it support registering tools, and attempting to do so will result in an error.

See `chat_snowflake()` to chat with more general-purpose models hosted on Snowflake.

Usage

```
chat_cortex_analyst(
  account = snowflake_account(),
  credentials = NULL,
  model_spec = NULL,
  model_file = NULL,
  api_args = list(),
  echo = c("none", "output", "all")
)
```

Arguments

`account` A Snowflake account identifier, e.g. `"testorg-test_account"`. Defaults to the value of the `SNOWFLAKE_ACCOUNT` environment variable.

`credentials` A list of authentication headers to pass into `httr2::req_headers()`, a function that returns them when called, or `NULL`, the default, to use ambient credentials.

`model_spec` A semantic model specification, or `NULL` when using `model_file` instead.

`model_file` Path to a semantic model file stored in a Snowflake Stage, or `NULL` when using `model_spec` instead.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
chat <- chat_cortex_analyst(
  model_file = "@my_db.my_schema.my_stage/model.yaml"
)
chat$chat("What questions can I ask?")
```

chat_databricks *Chat with a model hosted on Databricks*

Description

Databricks provides out-of-the-box access to a number of foundation models and can also serve as a gateway for external models hosted by a third party.

Authentication:

`chat_databricks()` picks up on ambient Databricks credentials for a subset of the Databricks client unified authentication model. Specifically, it supports:

- Personal access tokens
- Service principals via OAuth (OAuth M2M)
- User account via OAuth (OAuth U2M)
- Authentication via the Databricks CLI
- Posit Workbench-managed credentials
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Known limitations:

Databricks models do not support images, but they do support structured outputs. Tool calling support is also very limited at present and is currently not supported by ellmer.

Usage

```
chat_databricks(
  workspace = databricks_workspace(),
  system_prompt = NULL,
  model = NULL,
  token = NULL,
  api_args = list(),
  echo = c("none", "output", "all")
)
```

Arguments

`workspace` The URL of a Databricks workspace, e.g. `"https://example.cloud.databricks.com"`. Will use the value of the environment variable `DATABRICKS_HOST`, if set.

`system_prompt` A system prompt to set the behavior of the assistant.

`model` The model to use for the chat (defaults to "databricks-dbrx-instruct"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

Available foundational models include:

- `databricks-dbrx-instruct` (the default)
- `databricks-mixtral-8x7b-instruct`
- `databricks-meta-llama-3-1-70b-instruct`
- `databricks-meta-llama-3-1-405b-instruct`

`token` An authentication token for the Databricks workspace, or `NULL` to use ambient credentials.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

echo — One of the following options:

- none: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_databricks()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_deepseek *Chat with a model hosted on DeepSeek*

Description

Sign up at `https://platform.deepseek.com`.

Known limitations:

- Structured data extraction is not supported.
- Images are not supported.

Usage

```
chat_deepseek(
  system_prompt = NULL,
  base_url = "https://api.deepseek.com",
  api_key = deepseek_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`base_url` The base URL to the endpoint; the default uses DeepSeek.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `DEEPSEEK_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`model` The model to use for the chat (defaults to "deepseek-chat"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_deepseek()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_github *Chat with a model hosted on the GitHub model marketplace*

Description

GitHub (via Azure) hosts a number of open source and OpenAI models. To access the GitHub model marketplace, you will need to apply for and be accepted into the beta access program. See https://github.com/marketplace/models for details.

This function is a lightweight wrapper around `chat_openai()` with the defaults tweaked for the GitHub model marketplace.

Usage

```
chat_github(
  system_prompt = NULL,
  base_url = "https://models.inference.ai.azure.com/",
  api_key = github_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`api_key` The API key to use for authentication. You generally should not supply this directly, but instead manage your GitHub credentials as described in https://usethis.r-lib.org/articles/git-credentials.html. For headless environments, this will also look in the `GITHUB_PAT` env var.

`model` The model to use for the chat (defaults to "gpt-4o"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_github()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_google_gemini` *Chat with a Google Gemini or Vertex AI model*

Description

Google's AI offering is broken up into two parts: Gemini and Vertex AI. Most enterprises are likely to use Vertex AI, and individuals are likely to use Gemini.

Use `google_upload()` to upload files (PDFs, images, video, audio, etc.)

Authentication:

By default, `chat_google_gemini()` will use Google's default application credentials if there is no API key provided. This requires the **gargle** package.

It can also pick up on viewer-based credentials on Posit Connect. This in turn requires the **connectcreds** package.

Usage

```
chat_google_gemini(
  system_prompt = NULL,
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL
)

chat_google_vertex(
  location,
  project_id,
  system_prompt = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL
)

models_google_gemini(
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = NULL
)

models_google_vertex(location, project_id)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `GOOGLE_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. For Gemini, you can alternatively set `GEMINI_API_KEY`.

`model` The model to use for the chat (defaults to "gemini-2.0-flash"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_google_gemini()` to see all options.

`params` Common model parameters, usually created by `params()`.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

`location` Location, e.g. `us-east1`, `me-central1`, `africa-south1`.

`project_id` Project ID.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_cortex_analyst(), chat_databricks(), chat_deepseek(), chat_github(), chat_groq(), chat_huggingface(), chat_mistral(), chat_ollama(), chat_openai(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_google_gemini()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_groq *Chat with a model hosted on Groq*

Description

Sign up at https://groq.com.

This function is a lightweight wrapper around chat_openai() with the defaults tweaked for groq.

Known limitations:

groq does not currently support structured data extraction.

Usage

```
chat_groq(
  system_prompt = NULL,
  base_url = "https://api.groq.com/openai/v1",
  api_key = groq_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the endpoint; the default uses OpenAI.
api_key	API key to use for authentication. You generally should not supply this directly, but instead set the GROQ_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ().

`model` The model to use for the chat (defaults to "llama3-8b-8192"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_groq()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_huggingface` *Chat with a model hosted on Hugging Face Serverless Inference API*

Description

Hugging Face hosts a variety of open-source and proprietary AI models available via their Inference API. To use the Hugging Face API, you must have an Access Token, which you can obtain from your Hugging Face account (ensure that at least "Make calls to Inference Providers" and "Make calls to your Inference Endpoints" is checked).

This function is a lightweight wrapper around `chat_openai()`, with the defaults adjusted for Hugging Face.

Known limitations:

- Parameter support is hit or miss.
- Tool calling is currently broken in the API.
- While images are technically supported, I couldn't find any models that returned useful respones.
- Some models do not support the chat interface or parts of it, for example `google/gemma-2-2b-it` does not support a system prompt. You will need to carefully choose the model.

So overall, not something we could recommend at the moment.

Usage

```
chat_huggingface(
  system_prompt = NULL,
  params = NULL,
  api_key = hf_key(),
  model = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`params` Common model parameters, usually created by `params()`.

`api_key` The API key to use for authentication. You generally should not supply this directly, but instead set the `HUGGINGFACE_API_KEY` environment variable.

`model` The model to use for the chat (defaults to "meta-llama/Llama-3.1-8B-Instruct"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_huggingface()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_mistral` *Chat with a model hosted on Mistral's La Platforme*

Description

Get your API key from https://console.mistral.ai/api-keys.

Known limitations:

- Tool calling is unstable.
- Images require a model that supports images.

Usage

```
chat_mistral(
  system_prompt = NULL,
  params = NULL,
  api_key = mistral_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`params` Common model parameters, usually created by `params()`.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `MISTRAL_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`model` The model to use for the chat (defaults to "mistral-large-latest"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_mistral()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_ollama` *Chat with a local Ollama model*

Description

To use `chat_ollama()` first download and install Ollama. Then install some models either from the command line (e.g. with `ollama pull llama3.1`) or within R using ollamar (e.g. `ollamar::pull("llama3.1")`).

This function is a lightweight wrapper around `chat_openai()` with the defaults tweaked for ollama.

Known limitations:

- Tool calling is not supported with streaming (i.e. when `echo` is `"text"` or `"all"`)
- Models can only use 2048 input tokens, and there's no way to get them to use more, except by creating a custom model with a different default.
- Tool calling generally seems quite weak, at least with the models I have tried it with.

Usage

```
chat_ollama(
  system_prompt = NULL,
  base_url = "http://localhost:11434",
  model,
  seed = NULL,
  api_args = list(),
  echo = NULL,
  api_key = NULL
)

models_ollama(base_url = "http://localhost:11434")
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`base_url`	The base URL to the endpoint; the default uses OpenAI.
`model`	The model to use for the chat. Use `models_ollama()` to see all options.
`seed`	Optional integer seed that ChatGPT uses to try and make output more reproducible.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
`echo`	One of the following options: • `none`: don't emit any output (default when running in a function). • `text`: echo text output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
`api_key`	Ollama doesn't require an API key for local usage and in most cases you do not need to provide an `api_key`. However, if you're accessing an Ollama instance hosted behind a reverse proxy or secured endpoint that enforces bearer-token authentication, you can set `api_key` (or the `OLLAMA_API_KEY` environment variable).

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_ollama(model = "llama3.2")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_openai *Chat with an OpenAI model*

Description

OpenAI provides a number of chat-based models, mostly under the ChatGPT brand. Note that a ChatGPT Plus membership does not grant access to the API. You will need to sign up for a developer account (and pay for it) at the developer platform.

Usage

```
chat_openai(
  system_prompt = NULL,
  base_url = "https://api.openai.com/v1",
  api_key = openai_key(),
  model = NULL,
  params = NULL,
  seed = lifecycle::deprecated(),
  api_args = list(),
  echo = c("none", "output", "all")
)

models_openai(base_url = "https://api.openai.com/v1", api_key = openai_key())
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `OPENAI_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`model` The model to use for the chat (defaults to "gpt-4.1"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_openai()` to see all options.

`params` Common model parameters, usually created by `params()`.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_cortex_analyst()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
chat <- chat_openai()
chat$chat("
  What is the difference between a tibble and a data frame?
  Answer with a bulleted list
")

chat$chat("Tell me three funny jokes about statisticians")
```

`chat_openrouter` *Chat with one of the many models hosted on OpenRouter*

Description

Sign up at `https://openrouter.ai`.

Support for features depends on the underlying model that you use; see `https://openrouter.ai/models` for details.

Usage

```
chat_openrouter(
  system_prompt = NULL,
  api_key = openrouter_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = c("none", "output", "all")
)
```

Arguments

system_prompt A system prompt to set the behavior of the assistant.

api_key API key to use for authentication.

You generally should not supply this directly, but instead set the OPENROUTER_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ().

model The model to use for the chat (defaults to "gpt-4o"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

seed Optional integer seed that ChatGPT uses to try and make output more reproducible.

api_args Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().

echo One of the following options:

- none: don't emit any output (default when running in a function).
- text: echo text output as it streams in (default when running at the console).
- all: echo all input and output.

Note this only affects the chat() method.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_cortex_analyst(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_mistral(), chat_ollama(), chat_openai(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_openrouter()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_perplexity *Chat with a model hosted on perplexity.ai*

Description

Sign up at https://www.perplexity.ai.

Perplexity AI is a platform for running LLMs that are capable of searching the web in real-time to help them answer questions with information that may not have been available when the model was trained.

This function is a lightweight wrapper around chat_openai() with the defaults tweaked for Perplexity AI.

Usage

```
chat_perplexity(
  system_prompt = NULL,
  base_url = "https://api.perplexity.ai/",
  api_key = perplexity_key(),
  model = NULL,
  seed = NULL,
  api_args = list(),
  echo = NULL
)
```

Arguments

`system_prompt` A system prompt to set the behavior of the assistant.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `PERPLEXITY_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`model` The model to use for the chat (defaults to "llama-3.1-sonar-small-128k-online"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.

`seed` Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_cortex_analyst(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_mistral(), chat_ollama(), chat_openai(), chat_openrouter(), chat_portkey()

Examples

```
## Not run:
chat <- chat_perplexity()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_portkey *Chat with a model hosted on PortkeyAI*

Description

PortkeyAI provides an interface (AI Gateway) to connect through its Universal API to a variety of LLMs providers with a single endpoint.

Authentication:

API keys together with configurations of LLM providers are stored inside Portkey application.

Usage

```
chat_portkey(
  system_prompt = NULL,
  base_url = "https://api.portkey.ai/v1",
  api_key = portkeyai_key(),
  virtual_key = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL
)

models_portkey(
  base_url = "https://api.portkey.ai/v1",
  api_key = portkeyai_key(),
  virtual_key = NULL
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the endpoint; the default uses OpenAI.
api_key	API key to use for authentication. You generally should not supply this directly, but instead set the PORTKEY_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ().
virtual_key	A virtual identifier storing LLM provider's API key. See documentation.
model	The model to use for the chat (defaults to "gpt-4o"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use models_openai() to see all options.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • text: echo text output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_cortex_analyst(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_mistral(), chat_ollama(), chat_openai(), chat_openrouter(), chat_perplexity()

Examples

```
## Not run:
chat <- chat_portkey(virtual_key = Sys.getenv("PORTKEY_VIRTUAL_KEY"))
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_snowflake` *Chat with a model hosted on Snowflake*

Description

The Snowflake provider allows you to interact with LLM models available through the Cortex LLM REST API.

Authentication:

`chat_snowflake()` picks up the following ambient Snowflake credentials:

- A static OAuth token defined via the `SNOWFLAKE_TOKEN` environment variable.
- Key-pair authentication credentials defined via the `SNOWFLAKE_USER` and `SNOWFLAKE_PRIVATE_KEY` (which can be a PEM-encoded private key or a path to one) environment variables.
- Posit Workbench-managed Snowflake credentials for the corresponding `account`.
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Known limitations:

Note that Snowflake-hosted models do not support images, tool calling, or structured outputs.

See `chat_cortex_analyst()` to chat with the Snowflake Cortex Analyst rather than a general-purpose model.

Usage

```
chat_snowflake(
  system_prompt = NULL,
  account = snowflake_account(),
  credentials = NULL,
  model = NULL,
  api_args = list(),
  echo = c("none", "output", "all")
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`account`	A Snowflake account identifier, e.g. `"testorg-test_account"`. Defaults to the value of the `SNOWFLAKE_ACCOUNT` environment variable.
`credentials`	A list of authentication headers to pass into `httr2::req_headers()`, a function that returns them when called, or `NULL`, the default, to use ambient credentials.
`model`	The model to use for the chat (defaults to "llama3.1-70b"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
`echo`	One of the following options:

- none: don't emit any output (default when running in a function).
- text: echo text output as it streams in (default when running at the console).
- all: echo all input and output.

Note this only affects the chat() method.

Value

A Chat object.

Examples

```
chat <- chat_snowflake()
chat$chat("Tell me a joke in the form of a SQL query.")
```

chat_vllm *Chat with a model hosted by vLLM*

Description

vLLM is an open source library that provides an efficient and convenient LLMs model server. You can use chat_vllm() to connect to endpoints powered by vLLM.

Usage

```
chat_vllm(
  base_url,
  system_prompt = NULL,
  model,
  seed = NULL,
  api_args = list(),
  api_key = vllm_key(),
  echo = NULL
)

models_vllm(base_url, api_key = vllm_key())
```

Arguments

base_url	The base URL to the endpoint; the default uses OpenAI.
system_prompt	A system prompt to set the behavior of the assistant.
model	The model to use for the chat. Use models_vllm() to see all options.
seed	Optional integer seed that ChatGPT uses to try and make output more reproducible.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `VLLM_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `text`: echo text output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

Examples

```
## Not run:
chat <- chat_vllm("http://my-vllm.com")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`Content` *Content types received from and sent to a chatbot*

Description

Use these functions if you're writing a package that extends ellmer and need to customise methods for various types of content. For normal use, see `content_image_url()` and friends.

ellmer abstracts away differences in the way that different Providers represent various types of content, allowing you to more easily write code that works with any chatbot. This set of classes represents types of content that can be either sent to and received from a provider:

- `ContentText`: simple text (often in markdown format). This is the only type of content that can be streamed live as it's received.
- `ContentImageRemote` and `ContentImageInline`: images, either as a pointer to a remote URL or included inline in the object. See `content_image_file()` and friends for convenient ways to construct these objects.
- `ContentToolRequest`: a request to perform a tool call (sent by the assistant).
- `ContentToolResult`: the result of calling the tool (sent by the user). This object is automatically created from the value returned by calling the `tool()` function. Alternatively, expert users can return a `ContentToolResult` from a `tool()` function to include additional data or to customize the display of the result.

Usage

```
Content()

ContentText(text = stop("Required"))

ContentImage()

ContentImageRemote(url = stop("Required"), detail = "")

ContentImageInline(type = stop("Required"), data = NULL)

ContentToolRequest(
  id = stop("Required"),
  name = stop("Required"),
  arguments = list(),
  tool = NULL
)

ContentToolResult(value = NULL, error = NULL, extra = list(), request = NULL)

ContentThinking(thinking = stop("Required"), extra = list())

ContentPDF(type = stop("Required"), data = stop("Required"))
```

Arguments

`text`	A single string.
`url`	URL to a remote image.
`detail`	Not currently used.
`type`	MIME type of the image.
`data`	Base64 encoded image data.
`id`	Tool call id (used to associate a request and a result). Automatically managed by **ellmer**.
`name`	Function name
`arguments`	Named list of arguments to call the function with.
`tool`	ellmer automatically matches a tool request to the tools defined for the chatbot. If `NULL`, the request did not match a defined tool.
`value`	The results of calling the tool function, if it succeeded.
`error`	The error message, as a string, or the error condition thrown as a result of a failure when calling the tool function. Must be `NULL` when the tool call is successful.
`extra`	Additional data.
`request`	The ContentToolRequest associated with the tool result, automatically added by **ellmer** when evaluating the tool call.
`thinking`	The text of the thinking output.

Value

S7 objects that all inherit from `Content`

Examples

```
Content()
ContentText("Tell me a joke")
ContentImageRemote("https://www.r-project.org/Rlogo.png")
ContentToolRequest(id = "abc", name = "mean", arguments = list(x = 1:5))
```

`contents_text` *Format contents into a textual representation*

Description

[Experimental]

These generic functions can be use to convert Turn contents or Content objects into textual representations.

- `contents_text()` is the most minimal and only includes ContentText objects in the output.
- `contents_markdown()` returns the text content (which it assumes to be markdown and does not convert it) plus markdown representations of images and other content types.
- `contents_html()` returns the text content, converted from markdown to HTML with `commonmark::markdown_html()` plus HTML representations of images and other content types.

These content types will continue to grow and change as ellmer evolves to support more providers and as providers add more content types.

Usage

```
contents_text(content, ...)

contents_html(content, ...)

contents_markdown(content, ...)
```

Arguments

`content`	The Turn or Content object to be converted into text. `contents_markdown()` also accepts Chat instances to turn the entire conversation history into markdown text.
`...`	Additional arguments passed to methods.

Value

A string of text, markdown or HTML.

Examples

```
turns <- list(
  Turn("user", contents = list(
    ContentText("What's this image?"),
    content_image_url("https://placehold.co/200x200")
  )),
  Turn("assistant", "It's a placeholder image.")
)

lapply(turns, contents_text)
lapply(turns, contents_markdown)
if (rlang::is_installed("commonmark")) {
  contents_html(turns[[1]])
}
```

---

content_image_url *Encode images for chat input*

Description

These functions are used to prepare image URLs and files for input to the chatbot. The `content_image_url()` function is used to provide a URL to an image, while `content_image_file()` is used to provide the image data itself.

Usage

```
content_image_url(url, detail = c("auto", "low", "high"))

content_image_file(path, content_type = "auto", resize = "low")

content_image_plot(width = 768, height = 768)
```

Arguments

`url`	The URL of the image to include in the chat input. Can be a `data:` URL or a regular URL. Valid image types are PNG, JPEG, WebP, and non-animated GIF.
`detail`	The detail setting for this image. Can be `"auto"`, `"low"`, or `"high"`.
`path`	The path to the image file to include in the chat input. Valid file extensions are `.png`, `.jpeg`, `.jpg`, `.webp`, and (non-animated) `.gif`.
`content_type`	The content type of the image (e.g. `image/png`). If `"auto"`, the content type is inferred from the file extension.
`resize`	If `"low"`, resize images to fit within 512x512. If `"high"`, resize to fit within 2000x768 or 768x2000. (See the OpenAI docs for more on why these specific sizes are used.) If `"none"`, do not resize. You can also pass a custom string to resize the image to a specific size, e.g. `"200x200"` to resize to 200x200 pixels while preserving aspect ratio. Append >

to resize only if the image is larger than the specified size, and `!` to ignore aspect ratio (e.g. `"300x200>!"`).

All values other than none require the `magick` package.

`width, height`	Width and height in pixels.

Value

An input object suitable for including in the `...` parameter of the `chat()`, `stream()`, `chat_async()`, or `stream_async()` methods.

Examples

```
chat <- chat_openai(echo = TRUE)
chat$chat(
  "What do you see in these images?",
  content_image_url("https://www.r-project.org/Rlogo.png"),
  content_image_file(system.file("httr2.png", package = "ellmer"))
)


plot(waiting ~ eruptions, data = faithful)
chat <- chat_openai(echo = TRUE)
chat$chat(
  "Describe this plot in one paragraph, as suitable for inclusion in
   alt-text. You should briefly describe the plot type, the axes, and
   2-5 major visual patterns.",
   content_image_plot()
)
```

`content_pdf_file` *Encode PDFs content for chat input*

Description

These functions are used to prepare PDFs as input to the chatbot. The `content_pdf_url()` function is used to provide a URL to an PDF file, while `content_pdf_file()` is used to for local PDF files.

Not all providers support PDF input, so check the documentation for the provider you are using.

Usage

```
content_pdf_file(path)

content_pdf_url(url)
```

Arguments

`path, url`	Path or URL to a PDF file.

Value

A ContentPDF object

create_tool_def *Create metadata for a tool*

Description

In order to use a function as a tool in a chat, you need to craft the right call to tool(). This function helps you do that for documented functions by extracting the function's R documentation and creating a tool() call for you, using an LLM. It's meant to be used interactively while writing your code, not as part of your final code.

If the function has package documentation, that will be used. Otherwise, if the source code of the function can be automatically detected, then the comments immediately preceding the function are used (especially helpful if those are Roxygen comments). If neither are available, then just the function signature is used.

Note that this function is inherently imperfect. It can't handle all possible R functions, because not all parameters are suitable for use in a tool call (for example, because they're not serializable to simple JSON objects). The documentation might not specify the expected shape of arguments to the level of detail that would allow an exact JSON schema to be generated. Please be sure to review the generated code before using it!

Usage

```
create_tool_def(
  topic,
  chat = NULL,
  model = deprecated(),
  echo = interactive(),
  verbose = FALSE
)
```

Arguments

topic	A symbol or string literal naming the function to create metadata for. Can also be an expression of the form pkg::fun.
chat	A Chat object used to generate the output. If NULL (the default) uses chat_openai().
model	lifecycle::badge("deprecated") Formally used for definining the model used by the chat. Now supply chat instead.
echo	Emit the registration code to the console. Defaults to TRUE in interactive sessions.
verbose	If TRUE, print the input we send to the LLM, which may be useful for debugging unexpectedly poor results.

Value

A `register_tool` call that you can copy and paste into your code. Returned invisibly if echo is `TRUE`.

Examples

```
## Not run:
  # These are all equivalent
  create_tool_def(rnorm)
  create_tool_def(stats::rnorm)
  create_tool_def("rnorm")
  create_tool_def("rnorm", chat = chat_azure_openai())

## End(Not run)
```

`google_upload` *Upload a file to gemini*

Description

[Experimental]

This function uploads a file then waits for Gemini to finish processing it so that you can immediately use it in a prompt. It's experimental because it's currently Gemini specific, and we expect other providers to evolve similar feature in the future.

Uploaded files are automatically deleted after 2 days. Each file must be less than 2 GB and you can upload a total of 20 GB. ellmer doesn't currently provide a way to delete files early; please file an issue if this would be useful for you.

Usage

```
google_upload(
  path,
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = NULL,
  mime_type = NULL
)
```

Arguments

`path` Path to a file to upload.

`base_url` The base URL to the endpoint; the default uses OpenAI.

`api_key` API key to use for authentication.

You generally should not supply this directly, but instead set the `GOOGLE_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. For Gemini, you can alternatively set `GEMINI_API_KEY`.

mime_type Optionally, specify the mime type of the file. If not specified, will be guesses from the file extension.

Value

A <ContentUploaded> object that can be passed to $chat().

Examples

```
## Not run:
file <- google_upload("path/to/file.pdf")

chat <- chat_openai()
chat$chat(file, "Give me a three paragraph summary of this PDF")

## End(Not run)
```

interpolate *Helpers for interpolating data into prompts*

Description

These functions are lightweight wrappers around glue that make it easier to interpolate dynamic data into a static prompt:

- interpolate() works with a string.
- interpolate_file() works with a file.
- interpolate_package() works with a file in the insts/prompt directory of a package.

Compared to glue, dynamic values should be wrapped in {{ }}, making it easier to include R code and JSON in your prompt.

Usage

```
interpolate(prompt, ..., .envir = parent.frame())

interpolate_file(path, ..., .envir = parent.frame())

interpolate_package(package, path, ..., .envir = parent.frame())
```

Arguments

prompt A prompt string. You should not generally expose this to the end user, since glue interpolation makes it easy to run arbitrary code.

... Define additional temporary variables for substitution.

.envir Environment to evaluate ... expressions in. Used when wrapping in another function. See vignette("wrappers", package = "glue") for more details.

path A path to a prompt file (often a .md).

package Package name.

Value

A {glue} string.

Examples

```
joke <- "You're a cool dude who loves to make jokes. Tell me a joke about {{topic}}."

# You can supply valuese directly:
interpolate(joke, topic = "bananas")

# Or allow interpolate to find them in the current environment:
topic <- "applies"
interpolate(joke)
```

`live_console` *Open a live chat application*

Description

- `live_console()` lets you chat interactively in the console.
- `live_browser()` lets you chat interactively in a browser.

Note that these functions will mutate the input `chat` object as you chat because your turns will be appended to the history.

Usage

```
live_console(chat, quiet = FALSE)

live_browser(chat, quiet = FALSE)
```

Arguments

`chat`	A chat object created by `chat_openai()` or friends.
`quiet`	If `TRUE`, suppresses the initial message that explains how to use the console.

Value

(Invisibly) The input `chat`.

Examples

```
## Not run:
chat <- chat_anthropic()
live_console(chat)
live_browser(chat)

## End(Not run)
```

parallel_chat *Submit multiple chats in parallel*

Description

[Experimental]

If you have multiple prompts, you can submit them in parallel. This is typically considerably faster than submitting them in sequence, especially with Gemini and OpenAI.

If you're using `chat_openai()` or `chat_anthropic()` and you're willing to wait longer, you might want to use `batch_chat()` instead, as it comes with a 50% discount in return for taking up to 24 hours.

Usage

```
parallel_chat(chat, prompts, max_active = 10, rpm = 500)

parallel_chat_structured(
  chat,
  prompts,
  type,
  convert = TRUE,
  include_tokens = FALSE,
  include_cost = FALSE,
  max_active = 10,
  rpm = 500
)
```

Arguments

`chat` A base chat object.

`prompts` A vector created by `interpolate()` or a list of character vectors.

`max_active` The maximum number of simultaneous requests to send.

For `chat_anthropic()`, note that the number of active connections is limited primarily by the output tokens per minute limit (OTPM) which is estimated from the `max_tokens` parameter, which defaults to 4096. That means if your usage tier limits you to 16,000 OTPM, you should either set `max_active = 4` (16,000 / 4096) to decrease the number of active connections or use `params()` in `chat_anthropic()` to decrease `max_tokens`.

`rpm` Maximum number of requests per minute.

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`convert` If `TRUE`, automatically convert from JSON lists to R data types using the schema. This typically works best when `type` is `type_object()` as this will give you a data frame with one column for each property. If `FALSE`, returns a list.

`include_tokens` If `TRUE`, and the result is a data frame, will add `input_tokens` and `output_tokens` columns giving the total input and output tokens for each prompt.

`include_cost` If `TRUE`, and the result is a data frame, will add `cost` column giving the cost of each prompt.

Value

For `parallel_chat()`, a list of Chat objects, one for each prompt. For `parallel_chat_structured()`, a single structured data object with one element for each prompt. Typically, when `type` is an object, this will will be a data frame with one row for each prompt, and one column for each property.

Examples

```
chat <- chat_openai()

# Chat ----------------------------------------------------------------------
country <- c("Canada", "New Zealand", "Jamaica", "United States")
prompts <- interpolate("What's the capital of {{country}}?")
parallel_chat(chat, prompts)

# Structured data -----------------------------------------------------------
prompts <- list(
  "I go by Alex. 42 years on this planet and counting.",
  "Pleased to meet you! I'm Jamal, age 27.",
  "They call me Li Wei. Nineteen years young.",
  "Fatima here. Just celebrated my 35th birthday last week.",
  "The name's Robert - 51 years old and proud of it.",
  "Kwame here - just hit the big 5-0 this year."
)
type_person <- type_object(name = type_string(), age = type_number())
parallel_chat_structured(chat, prompts, type_person)
```

`params` *Standard model parameters*

Description

This helper function makes it easier to create a list of parameters used across many models. The parameter names are automatically standardised and included in the correctly place in the API call.

Note that parameters that are not supported by a given provider will generate a warning, not an error. This allows you to use the same set of parameters across multiple providers.

Usage

```
params(
  temperature = NULL,
  top_p = NULL,
  top_k = NULL,
  frequency_penalty = NULL,
  presence_penalty = NULL,
  seed = NULL,
  max_tokens = NULL,
  log_probs = NULL,
  stop_sequences = NULL,
  ...
)
```

Arguments

`temperature`	Temperature of the sampling distribution.
`top_p`	The cumulative probability for token selection.
`top_k`	The number of highest probability vocabulary tokens to keep.
`frequency_penalty`	Frequency penalty for generated tokens.
`presence_penalty`	Presence penalty for generated tokens.
`seed`	Seed for random number generator.
`max_tokens`	Maximum number of tokens to generate.
`log_probs`	Include the log probabilities in the output?
`stop_sequences`	A character vector of tokens to stop generation on.
`...`	Additional named parameters to send to the provider.

`Provider` *A chatbot provider*

Description

A Provider captures the details of one chatbot service/API. This captures how the API works, not the details of the underlying large language model. Different providers might offer the same (open source) model behind a different API.

Usage

```
Provider(
  name = stop("Required"),
  model = stop("Required"),
  base_url = stop("Required"),
  params = list(),
  extra_args = list()
)
```

Arguments

`name` Name of the provider.

`model` Name of the model.

`base_url` The base URL for the API.

`params` A list of standard parameters created by `params()`.

`extra_args` Arbitrary extra arguments to be included in the request body.

Details

To add support for a new backend, you will need to subclass `Provider` (adding any additional fields that your provider needs) and then implement the various generics that control the behavior of each provider.

Value

An S7 Provider object.

Examples

```
Provider(
  name = "CoolModels",
  model = "my_model",
  base_url = "https://cool-models.com"
)
```

`token_usage` *Report on token usage in the current session*

Description

Call this function to find out the cumulative number of tokens that you have sent and recieved in the current session. The price will be shown if known.

Usage

```
token_usage()
```

Value

A data frame

Examples

```
token_usage()
```

tool Define a tool

Description

Define an R function for use by a chatbot. The function will always be run in the current R instance.

Learn more in vignette("tool-calling").

Usage

```
tool(
  .fun,
  .description,
  ...,
  .name = NULL,
  .convert = TRUE,
  .annotations = list()
)
```

Arguments

.fun	The function to be invoked when the tool is called. The return value of the function is sent back to the chatbot. Expert users can customize the tool result by returning a ContentToolResult object.
.description	A detailed description of what the function does. Generally, the more information that you can provide here, the better.
...	Name-type pairs that define the arguments accepted by the function. Each element should be created by a type_*() function.
.name	The name of the function.
.convert	Should JSON inputs be automatically convert to their R data type equivalents? Defaults to TRUE.
.annotations	Additional properties that describe the tool and its behavior. Usually created by tool_annotations(), where you can find a description of the annotation properties recommended by the Model Context Protocol.

Value

An S7 ToolDef object.

See Also

Other tool calling helpers: tool_annotations(), tool_reject()

Examples

```
# First define the metadata that the model uses to figure out when to
# call the tool
tool_rnorm <- tool(
  rnorm,
  "Drawn numbers from a random normal distribution",
  n = type_integer("The number of observations. Must be a positive integer."),
  mean = type_number("The mean value of the distribution."),
  sd = type_number("The standard deviation of the distribution. Must be a non-negative number."),
  .annotations = tool_annotations(
    title = "Draw Random Normal Numbers",
    read_only_hint = TRUE,
    open_world_hint = FALSE
  )
)
chat <- chat_openai()
# Then register it
chat$register_tool(tool_rnorm)

# Then ask a question that needs it.
chat$chat("
  Give me five numbers from a random normal distribution.
")

# Look at the chat history to see how tool calling works:
# Assistant sends a tool request which is evaluated locally and
# results are send back in a tool result.
```

tool_annotations *Tool annotations*

Description

Tool annotations are additional properties that, when passed to the `.annotations` argument of `tool()`, provide additional information about the tool and its behavior. This information can be used for display to users, for example in a Shiny app or another user interface.

The annotations in `tool_annotations()` are drawn from the Model Context Protocol and are considered *hints*. Tool authors should use these annotations to communicate tool properties, but users should note that these annotations are not guaranteed.

Usage

```
tool_annotations(
  title = NULL,
  read_only_hint = NULL,
  open_world_hint = NULL,
  idempotent_hint = NULL,
  destructive_hint = NULL,
  ...
)
```

Arguments

`title` A human-readable title for the tool.

`read_only_hint` If TRUE, the tool does not modify its environment.

`open_world_hint` If TRUE, the tool may interact with an "open world" of external entities. If FALSE, the tool's domain of interaction is closed. For example, the world of a web search tool is open, but the world of a memory tool is not.

`idempotent_hint` If TRUE, calling the tool repeatedly with the same arguments will have no additional effect on its environment. (Only meaningful when read_only_hint is FALSE.)

`destructive_hint` If TRUE, the tool may perform destructive updates to its environment, otherwise it only performs additive updates. (Only meaningful when read_only_hint is FALSE.)

`...` Additional named parameters to include in the tool annotations.

Value

A list of tool annotations.

See Also

Other tool calling helpers: tool(), tool_reject()

Examples

```
# See ?tool() for a full example using this function.
# We're creating a tool around R's `rnorm()` function to allow the chatbot to
# generate random numbers from a normal distribution.
tool_rnorm <- tool(
  rnorm,
  # Describe the tool function to the LLM
  .description = "Drawn numbers from a random normal distribution",
  # Describe the parameters used by the tool function
  n = type_integer("The number of observations. Must be a positive integer."),
  mean = type_number("The mean value of the distribution."),
 sd = type_number("The standard deviation of the distribution. Must be a non-negative number."),
  # Tool annotations optionally provide additional context to the LLM
  .annotations = tool_annotations(
    title = "Draw Random Normal Numbers",
    read_only_hint = TRUE, # the tool does not modify any state
    open_world_hint = FALSE # the tool does not interact with the outside world
  )
)
```

`tool_reject` *Reject a tool call*

Description

Throws an error to reject a tool call. `tool_reject()` can be used within the tool function to indicate that the tool call should not be processed. `tool_reject()` can also be called in an `Chat$on_tool_request()` callback. When used in the callback, the tool call is rejected before the tool function is invoked.

Here's an example where `utils::askYesNo()` is used to ask the user for permission before accessing their current working directory. This happens directly in the tool function and is appropriate when you write the tool definition and know exactly how it will be called.

```
chat <- chat_openai(model = "gpt-4.1-nano")

list_files <- function() {
  allow_read <- utils::askYesNo(
    "Would you like to allow access to your current directory?"
  )
  if (isTRUE(allow_read)) {
    dir(pattern = "[.](r|R|csv)$")
  } else {
    tool_reject()
  }
}

chat$register_tool(tool(
  list_files,
  "List files in the user's current directory"
))

chat$chat("What files are available in my current directory?")
#> [tool call] list_files()
#> Would you like to allow access to your current directory? (Yes/no/cancel) no
#> #> Error: Tool call rejected. The user has chosen to disallow the tool #' call.
#> It seems I am unable to access the files in your current directory right now.
#> If you can tell me what specific files you're looking for or if you can #' provide
#> the list, I can assist you further.

chat$chat("Try again.")
#> [tool call] list_files()
#> Would you like to allow access to your current directory? (Yes/no/cancel) yes
#> #> app.R
#> #> data.csv
#> The files available in your current directory are "app.R" and "data.csv".
```

You can achieve a similar experience with tools written by others by using a `tool_request` callback. In the next example, imagine the tool is provided by a third-party package. This example implements a simple menu to ask the user for consent before running *any* tool.

```
packaged_list_files_tool <- tool(
  function() dir(pattern = "[.](r|R|csv)$"),
  "List files in the user's current directory"
)

chat <- chat_openai(model = "gpt-4.1-nano")
chat$register_tool(packaged_list_files_tool)

always_allowed <- c()

# ContentToolRequest
chat$on_tool_request(function(request) {
  if (request@name %in% always_allowed) return()

  answer <- utils::menu(
    title = sprintf("Allow tool `%s()` to run?", request@name),
    choices = c("Always", "Once", "No"),
    graphics = FALSE
  )

  if (answer == 1) {
    always_allowed <<- append(always_allowed, request@name)
  } else if (answer %in% c(0, 3)) {
    tool_reject()
  }
})

# Try choosing different answers to the menu each time
chat$chat("What files are available in my current directory?")
chat$chat("How about now?")
chat$chat("And again now?")
```

Usage

```
tool_reject(reason = "The user has chosen to disallow the tool call.")
```

Arguments

`reason`	A character string describing the reason for rejecting the tool call.

Value

Throws an error of class `ellmer_tool_reject` with the provided reason.

See Also

Other tool calling helpers: `tool()`, `tool_annotations()`

`Turn` *A user or assistant turn*

Description

Every conversation with a chatbot consists of pairs of user and assistant turns, corresponding to an HTTP request and response. These turns are represented by the `Turn` object, which contains a list of Contents representing the individual messages within the turn. These might be text, images, tool requests (assistant only), or tool responses (user only).

Note that a call to `$chat()` and related functions may result in multiple user-assistant turn cycles. For example, if you have registered tools, ellmer will automatically handle the tool calling loop, which may result in any number of additional cycles. Learn more about tool calling in `vignette("tool-calling")`.

Usage

```
Turn(role, contents = list(), json = list(), tokens = c(0, 0))
```

Arguments

`role`	Either "user", "assistant", or "system".
`contents`	A list of Content objects.
`json`	The serialized JSON corresponding to the underlying data of the turns. Currently only provided for assistant. This is useful if there's information returned by the provider that ellmer doesn't otherwise expose.
`tokens`	A numeric vector of length 2 representing the number of input and output tokens (respectively) used in this turn. Currently only recorded for assistant turns.

Value

An S7 `Turn` object

Examples

```
Turn(role = "user", contents = list(ContentText("Hello, world!")))
```

Type *Type definitions for function calling and structured data extraction.*

Description

These S7 classes are provided for use by package devlopers who are extending ellmer. In every day use, use `type_boolean()` and friends.

Usage

```
TypeBasic(description = NULL, required = TRUE, type = stop("Required"))

TypeEnum(description = NULL, required = TRUE, values = character(0))

TypeArray(description = NULL, required = TRUE, items = Type())

TypeObject(
  description = NULL,
  required = TRUE,
  properties = list(),
  additional_properties = TRUE
)
```

Arguments

`description` The purpose of the component. This is used by the LLM to determine what values to pass to the tool or what values to extract in the structured data, so the more detail that you can provide here, the better.

`required` Is the component or argument required?

In type descriptions for structured data, if `required = FALSE` and the component does not exist in the data, the LLM may hallucinate a value. Only applies when the element is nested inside of a `type_object()`.

In tool definitions, `required = TRUE` signals that the LLM should always provide a value. Arguments with `required = FALSE` should have a default value in the tool function's definition. If the LLM does not provide a value, the default value will be used.

`type` Basic type name. Must be one of `boolean`, `integer`, `number`, or `string`.

`values` Character vector of permitted values.

`items` The type of the array items. Can be created by any of the `type_` function.

`properties` Named list of properties stored inside the object. Each element should be an S7 Type object.‘

`additional_properties`
Can the object have arbitrary additional properties that are not explicitly listed? Only supported by Claude.

Value

S7 objects inheriting from Type

Examples

```
TypeBasic(type = "boolean")
TypeArray(items = TypeBasic(type = "boolean"))
```

`type_boolean` *Type specifications*

Description

These functions specify object types in a way that chatbots understand and are used for tool calling and structured data extraction. Their names are based on the JSON schema, which is what the APIs expect behind the scenes. The translation from R concepts to these types is fairly straightforward.

- `type_boolean()`, `type_integer()`, `type_number()`, and `type_string()` each represent scalars. These are equivalent to length-1 logical, integer, double, and character vectors (respectively).
- `type_enum()` is equivalent to a length-1 factor; it is a string that can only take the specified values.
- `type_array()` is equivalent to a vector in R. You can use it to represent an atomic vector: e.g. `type_array(items = type_boolean())` is equivalent to a logical vector and `type_array(items = type_string())` is equivalent to a character vector). You can also use it to represent a list of more complicated types where every element is the same type (R has no base equivalent to this), e.g. `type_array(items = type_array(items = type_string()))` represents a list of character vectors.
- `type_object()` is equivalent to a named list in R, but where every element must have the specified type. For example, `type_object(a = type_string(), b = type_array(type_integer()))` is equivalent to a list with an element called `a` that is a string and an element called `b` that is an integer vector.

Usage

```
type_boolean(description = NULL, required = TRUE)

type_integer(description = NULL, required = TRUE)

type_number(description = NULL, required = TRUE)

type_string(description = NULL, required = TRUE)

type_enum(description = NULL, values, required = TRUE)

type_array(description = NULL, items, required = TRUE)
```

```
type_object(
  .description = NULL,
  ...,
  .required = TRUE,
  .additional_properties = FALSE
)
```

Arguments

`description`, `.description`

The purpose of the component. This is used by the LLM to determine what values to pass to the tool or what values to extract in the structured data, so the more detail that you can provide here, the better.

`required`, `.required`

Is the component or argument required?

In type descriptions for structured data, if `required = FALSE` and the component does not exist in the data, the LLM may hallucinate a value. Only applies when the element is nested inside of a `type_object()`.

In tool definitions, `required = TRUE` signals that the LLM should always provide a value. Arguments with `required = FALSE` should have a default value in the tool function's definition. If the LLM does not provide a value, the default value will be used.

`values` Character vector of permitted values.

`items` The type of the array items. Can be created by any of the `type_` function.

`...` Name-type pairs defineing the components that the object must possess.

`.additional_properties`

Can the object have arbitrary additional properties that are not explicitly listed? Only supported by Claude.

Examples

```
# An integer vector
type_array(items = type_integer())

# The closest equivalent to a data frame is an array of objects
type_array(items = type_object(
   x = type_boolean(),
   y = type_string(),
   z = type_number()
))

# There's no specific type for dates, but you use a string with the
# requested format in the description (it's not gauranteed that you'll
# get this format back, but you should most of the time)
type_string("The creation date, in YYYY-MM-DD format.")
type_string("The update date, in dd/mm/yyyy format.")
```

Index