Package: ellmer (via r-universe)

January 9, 2025

Title Chat with Large Language Models

Version 0.1.0

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

Imports cli, coro (>= 1.1.0), glue, httr2 (>= 1.0.7), jsonlite, later (>= 1.4.0), promises (>= 1.3.1), R6, rlang (>= 1.1.0), S7 (>= 0.2.0)

Suggests base64enc, bslib, curl (>= 6.0.1), gitcreds, knitr, magick, openssl, paws.common, rmarkdown, shiny, shinychat (>= 0.1.1), 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' 'content.R' 'provider.R' 'as-json.R' 'utils-coro.R' 'chat.R' 'content-image.R' 'content-tools.R' 'ellmer-package.R' 'httr2.R' 'import-standalone-obj-type.R' 'import-standalone-purrr.R' 'import-standalone-types-check.R' 'interpolate.R' 'tools-def.R' 'turns.R' 'provider-openai.R' 'provider-azure.R' 'provider-bedrock.R' 'provider-claude.R' 'provider-cortex.R' 'provider-databricks.R' 'provider-gemini.R' 'provider-github.R' 'provider-groq.R' 'provider-ollama.R'

'provider-perplexity.R' 'provider-vllm.R' 'shiny.R' 'tokens.R'
'tools-def-auto.R' 'utils-cat.R' 'utils-merge.R' 'utils.R'
'zzz.R'

NeedsCompilation no

Author Hadley Wickham [aut, cre], Joe Cheng [aut], Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2025-01-09 13:50:05 UTC

Config/pak/sysreqs libssl-dev

Contents

Chat *A chat*

Description

A Chat is an sequence of 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$set_turns()
- Chat$get_system_prompt()
- Chat$set_system_prompt()
- Chat$tokens()
- Chat$last_turn()
- Chat$chat()
- Chat$extract_data()
- Chat$extract_data_async()
- Chat$chat_async()
- Chat$stream()
- Chat$stream_async()
- Chat$register_tool()
- Chat$clone()

Method new():

Usage:

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

Arguments:

provider A provider object.

turns An unnamed list of turns to start the chat with (i.e., continuing a previous conversation). If NULL or zero-length list, the conversation begins from scratch.

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

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

Usage:

```
Chat$get_system_prompt()
```

Method `set_system_prompt()`**:** Update the system prompt

Usage:

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

Arguments:

`value` A string giving the new system prompt

Method `tokens()`**:** List the number of tokens consumed by each assistant turn. Currently tokens are recorded for assistant turns only; so user turns will have zeros.

Usage:

```
Chat$tokens()
```

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 `extract_data()`**:** Extract structured data

Usage:

```
Chat$extract_data(..., 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 `extract_data_async()`**:** Extract structured data, asynchronously. Returns a promise that resolves to an object matching the type specification.

Usage:

```
Chat$extract_data_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(...)
```

Arguments:

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

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(...)
```

Arguments:

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

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(...)
```

Arguments:

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

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 `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_azure` *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.

Usage

```
chat_azure(
  endpoint = azure_endpoint(),
  deployment_id,
  api_version = NULL,
  system_prompt = NULL,
  turns = NULL,
  api_key = azure_key(),
  token = NULL,
  api_args = list(),
  echo = c("none", "text", "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.
api_version	The API version to use.
system_prompt	A system prompt to set the behavior of the assistant.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
api_key	The API key to use for authentication. You generally should not supply this directly, but instead set the `AZURE_OPENAI_API_KEY` environment variable.
token	Azure token for authentication. This is typically not required for Azure OpenAI API calls, but can be used if your setup requires it.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call.
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_azure(deployment_id = "gpt-4o-mini")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_bedrock` *Chat with an AWS bedrock model*

Description

AWS Bedrock provides a number of chat based models, including those Anthropic's Claude.

Authenthication is handled through {paws.common}, so if authenthication 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_bedrock(
  system_prompt = NULL,
  turns = NULL,
  model = NULL,
  profile = NULL,
  echo = NULL
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`turns`	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
`model`	The model to use for the chat. The default, `NULL`, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most casual use.
`profile`	AWS profile to use.
`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_claude()`, `chat_cortex()`, `chat_databricks()`, `chat_gemini()`, `chat_github()`, `chat_groq()`, `chat_ollama()`, `chat_openai()`, `chat_perplexity()`

Examples

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

## End(Not run)
```

`chat_claude` *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

To authenticate, we recommend saving your API key to the `ANTHROPIC_API_KEY` env var in your `.Renviron` (which you can easily edit by calling `usethis::edit_r_environ()`).

Usage

```
chat_claude(
  system_prompt = NULL,
  turns = NULL,
  max_tokens = 4096,
  model = NULL,
  api_args = list(),
  base_url = "https://api.anthropic.com/v1",
  api_key = anthropic_key(),
  echo = NULL
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`turns`	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
`max_tokens`	Maximum number of tokens to generate before stopping.
`model`	The model to use for the chat. The default, `NULL`, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most casual use.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call.
`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 set the `ANTHROPIC_API_KEY` environment variable.

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_bedrock(), chat_cortex(), chat_databricks(), chat_gemini(), chat_github(), chat_groq(), chat_ollama(), chat_openai(), chat_perplexity()

Examples

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

---

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

Description

Chat with the LLM-powered Snowflake Cortex Analyst.

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.

Authentication:

chat_cortex() 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.

Usage

```
chat_cortex(
  account = Sys.getenv("SNOWFLAKE_ACCOUNT"),
  credentials = NULL,
  model_spec = NULL,
  model_file = NULL,
  api_args = list(),
  echo = c("none", "text", "all")
)
```

Arguments

account	A Snowflake account identifier, e.g. "testorg-test_account".
credentials	A list of authentication headers to pass into httr2::req_headers(), a function that returns them when passed account as a parameter, or NULL 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.
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_bedrock(), chat_claude(), chat_databricks(), chat_gemini(), chat_github(), chat_groq(), chat_ollama(), chat_openai(), chat_perplexity()

Examples

```
chat <- chat_cortex(
  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.

Databricks models do not support images, but they do support structured outputs. Tool calling support is also very limited at present; too limited for `ellmer`'s tool calling features to work properly at all.

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

Usage

```
chat_databricks(
  workspace = databricks_workspace(),
  system_prompt = NULL,
  turns = NULL,
  model = NULL,
  token = NULL,
  api_args = list(),
  echo = c("none", "text", "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.
`turns`	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
`model`	The model to use for the chat. The default, `NULL`, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.

`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_bedrock()`, `chat_claude()`, `chat_cortex()`, `chat_gemini()`, `chat_github()`, `chat_groq()`, `chat_ollama()`, `chat_openai()`, `chat_perplexity()`

Examples

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

## End(Not run)
```

---

chat_gemini *Chat with a Google Gemini model*

Description

To authenticate, we recommend saving your API key to the `GOOGLE_API_KEY` env var in your `.Renviron` (which you can easily edit by calling `usethis::edit_r_environ()`).

Usage

```
chat_gemini(
  system_prompt = NULL,
  turns = NULL,
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = gemini_key(),
  model = NULL,
```

```
  api_args = list(),
  echo = NULL
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
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 set the GOOGLE_API_KEY environment variable.
model	The model to use for the chat. The default, NULL, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most casual use.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call.
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_bedrock(), chat_claude(), chat_cortex(), chat_databricks(), chat_github(), chat_groq(), chat_ollama(), chat_openai(), chat_perplexity()

Examples

```
## Not run:
chat <- chat_gemini()
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,
  turns = 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.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
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. The default, NULL, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
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_bedrock()`, `chat_claude()`, `chat_cortex()`, `chat_databricks()`, `chat_gemini()`, `chat_groq()`, `chat_ollama()`, `chat_openai()`, `chat_perplexity()`

Examples

```
## Not run:
chat <- chat_github()
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.

It does not currently support structured data extraction.

Usage

```
chat_groq(
  system_prompt = NULL,
  turns = 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.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
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 set the OPENAI_API_KEY environment variable.
model	The model to use for the chat. The default, NULL, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
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_bedrock(), chat_claude(), chat_cortex(), chat_databricks(), chat_gemini(), chat_github(), chat_ollama(), chat_openai(), chat_perplexity()

Examples

```
## Not run:
chat <- chat_groq()
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 from the command line, e.g. with `ollama pull llama3.1` or `ollama pull gemma2`.

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"`)
- Tool calling generally seems quite weak, at least with the models I have tried it with.

Usage

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

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`turns`	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
`base_url`	The base URL to the endpoint; the default uses OpenAI.
`model`	The model to use for the chat. The default, `NULL`, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
`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_bedrock(), chat_claude(), chat_cortex(), chat_databricks(), chat_gemini(), chat_github(), chat_groq(), chat_openai(), chat_perplexity()

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.

For authentication, we recommend saving your API key to the OPENAI_API_KEY environment variable in your .Renviron file. You can easily edit this file by calling usethis::edit_r_environ().

Usage

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

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
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 set the OPENAI_API_KEY environment variable.
model	The model to use for the chat. The default, NULL, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
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_bedrock(), chat_claude(), chat_cortex(), chat_databricks(), chat_gemini(), chat_github(), chat_groq(), chat_ollama(), chat_perplexity()

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 statistcians")
```

---

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,
  turns = 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.
turns	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
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 set the PERPLEXITY_API_KEY environment variable.
model	The model to use for the chat. The default, NULL, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
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_bedrock(), chat_claude(), chat_cortex(), chat_databricks(), chat_gemini(), chat_github(), chat_groq(), chat_ollama(), chat_openai()

Examples

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

## End(Not run)
```

`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,
  turns = NULL,
  model,
  seed = NULL,
  api_args = list(),
  api_key = vllm_key(),
  echo = NULL
)
```

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.
`turns`	A list of Turns to start the chat with (i.e., continuing a previous conversation). If not provided, the conversation begins from scratch.
`model`	The model to use for the chat. The default, `NULL`, will pick a reasonable default, and tell you about. We strongly recommend explicitly choosing a model for all but the most 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.
`api_key`	The API key to use for authentication. You generally should not supply this directly, but instead set the `VLLM_API_KEY` environment variable.
`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).

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()
)

ContentToolResult(id = stop("Required"), value = NULL, error = NULL)
```

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)
`name`	Function name
`arguments`	Named list of arguments to call the function with.
`value, error`	Either the results of calling the function if it succeeded, otherwise the error message, as a string. One of `value` and `error` will always be `NULL`.

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

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.

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 image content 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()
```

```
)
```

`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, model = "gpt-4o", 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`.
`model`	The OpenAI model to use for generating the metadata. Defaults to "gpt-4o".
`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")

## 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. Compared to glue, these functions expect you to wrap dynamic values 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())
```

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`).

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_claude()
live_console(chat)
live_browser(chat)

## End(Not run)
```

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(base_url = stop("Required"), extra_args = list())
```

Arguments

base_url	The base URL for the API.
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(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.

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)
```

Arguments

`.fun`	The function to be invoked when the tool is called.
`.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.

Value

An S7 `ToolDef` object.

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.")
)
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.
```

`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 required? If `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()`.
`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 required? If `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()`.

`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