Package 'shinychat'

May 9, 2026

Title Chat UI Component for 'shiny'

Version 0.3.0

Description Provides a scrolling chat interface with multiline input, suitable for creating chatbot apps based on Large Language Models (LLMs). Designed to work particularly well with the 'ellmer' R package for calling LLMs.

License MIT + file LICENSE

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

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

Imports base64enc, bslib, cli, coro, ellmer (>= 0.3.0), fastmap, htmltools, jsonlite, lifecycle, promises (>= 1.3.2), rlang (>= 1.1.0), S7, shiny (>= 1.10.0)

Suggests knitr, later, testthat (>= 3.0.0), withr

Config/Needs/website tidyverse/tidytemplate, rmarkdown, weathR, gt, glue, bsicons

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Author Joe Cheng [aut], Carson Sievert [aut], Garrick Aden-Buie [aut, cre] (ORCID: <https://orcid.org/0000-0002-7111-0077>), Barret Schloerke [aut] (ORCID: <https://orcid.org/0000-0001-9986-114X>), Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Garrick Aden-Buie <garrick@adenbuie.com>

Repository CRAN

Date/Publication 2025-11-20 14:00:02 UTC

Contents

`chat_app` *Open a live chat application in the browser*

Description

Create a simple Shiny app for live chatting using an ellmer::Chat object. Note that these functions will mutate the input `client` object as you chat because your turns will be appended to the history.

The app created by `chat_app()` is suitable for interactive use by a single user. For multi-user Shiny apps, use the Shiny module chat functions – `chat_mod_ui()` and `chat_mod_server()` – and be sure to create a new chat client for each user session.

Usage

```
chat_app(client, ..., bookmark_store = "url")

chat_mod_ui(id, ..., client = deprecated(), messages = NULL)

chat_mod_server(
  id,
  client,
  bookmark_on_input = TRUE,
  bookmark_on_response = TRUE
)
```

Arguments

`client`	A chat object created by **ellmer**, e.g. `ellmer::chat_openai()` and friends. This argument is deprecated in `chat_mod_ui()` because the client state is now managed by `chat_mod_server()`.
`...`	In `chat_app()`, additional arguments are passed to `shiny::shinyApp()`. In `chat_mod_ui()`, additional arguments are passed to `chat_ui()`.

bookmark_store The bookmarking store to use for the app. Passed to enable_bookmarking in shiny::shinyApp(). Defaults to "url", which uses the URL to store the chat state. URL-based bookmarking is limited in size; use "server" to store the state on the server side without size limitations; or disable bookmarking by setting this to "disable".

id The chat module ID.

messages Initial messages shown in the chat, used only when client (in chat_mod_ui()) doesn't already contain turns. Passed to messages in chat_ui().

bookmark_on_input A logical value determines if the bookmark should be updated when the user submits a message. Default is TRUE.

bookmark_on_response A logical value determines if the bookmark should be updated when the response stream completes. Default is TRUE.

Value

- chat_app() returns a shiny::shinyApp() object.
- chat_mod_ui() returns the UI for a shinychat module.
- chat_mod_server() includes the shinychat module server logic, and returns a list containing:
  - last_input: A reactive value containing the last user input.
  - last_turn: A reactive value containing the last assistant turn.
  - update_user_input(): A function to update the chat input or submit a new user input. Takes the same arguments as update_chat_user_input(), except for id and session, which are supplied automatically.
  - append(): A function to append a new message to the chat UI. Takes the same arguments as chat_append(), except for id and session, which are supplied automatically.
  - clear(): A function to clear the chat history and the chat UI. clear() takes an optional list of messages used to initialize the chat after clearing. messages should be a list of messages, where each message is a list with role and content fields. The client_history argument controls how the chat client's history is updated after clearing. It can be one of: "clear" the chat history; "set" the chat history to messages; "append" messages to the existing chat history; or "keep" the existing chat history.
  - client: The chat client object, which is mutated as you chat.

Functions

- chat_app(): A simple Shiny app for live chatting. Note that this app is suitable for interactive use by a single user; do not use chat_app() in a multi-user Shiny app context.
- chat_mod_ui(): A simple chat app module UI.
- chat_mod_server(): A simple chat app module server.

Examples

```
## Not run:
# Interactive in the console ----
```

```
client <- ellmer::chat_anthropic()
chat_app(client)

# Inside a Shiny app ----
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  titlePanel("shinychat example"),

  layout_columns(
    card(
      card_header("Chat with Claude"),
      chat_mod_ui(
        "claude",
        messages = list(
          "Hi! Use this chat interface to chat with Anthropic's `claude-3-5-sonnet`."
        )
      )
    ),
    card(
      card_header("Chat with ChatGPT"),
      chat_mod_ui(
        "openai",
        messages = list(
          "Hi! Use this chat interface to chat with OpenAI's `gpt-4o`."
        )
      )
    )
  )
)

server <- function(input, output, session) {
 claude <- ellmer::chat_anthropic(model = "claude-3-5-sonnet-latest") # Requires ANTHROPIC_API_KEY
  openai <- ellmer::chat_openai(model = "gpt-4o") # Requires OPENAI_API_KEY

  chat_mod_server("claude", claude)
  chat_mod_server("openai", openai)
}

shinyApp(ui, server)

## End(Not run)
```

---

chat_append *Append an assistant response (or user message) to a chat control*

Description

The `chat_append` function appends a message to an existing `chat_ui()`. The `response` can be a string, string generator, string promise, or string promise generator (as returned by the 'ellmer' package's `chat`, `stream`, `chat_async`, and `stream_async` methods, respectively).

This function should be called from a Shiny app's server. It is generally used to append the client's response to the chat, while user messages are added to the chat UI automatically by the front-end. You'd only need to use `chat_append(role="user")` if you are programmatically generating queries from the server and sending them on behalf of the user, and want them to be reflected in the UI.

Usage

```
chat_append(
  id,
  response,
  role = c("assistant", "user"),
  icon = NULL,
  session = getDefaultReactiveDomain()
)
```

Arguments

`id`	The ID of the chat element

`response`	The message or message stream to append to the chat element. The actual message content can one of the following:
	• A string, which is interpreted as markdown and rendered to HTML on the client.
	  – To prevent interpreting as markdown, mark the string as `htmltools::HTML()`.
	• A UI element.
	  – This includes `htmltools::tagList()`, which take UI elements (including strings) as children. In this case, strings are still interpreted as markdown as long as they're not inside HTML.

`role`	The role of the message (either "assistant" or "user"). Defaults to "assistant".
`icon`	An optional icon to display next to the message, currently only used for assistant messages. The icon can be any HTML element (e.g., an `htmltools::img()` tag) or a string of HTML.
`session`	The Shiny session object

Value

Returns a promise that resolves to the contents of the stream, or an error. This promise resolves when the message has been successfully sent to the client; note that it does not guarantee that the message was actually received or rendered by the client. The promise rejects if an error occurs while processing the response (see the "Error handling" section).

Error handling

If the response argument is a generator, promise, or promise generator, and an error occurs while producing the message (e.g., an iteration in stream_async fails), the promise returned by chat_append will reject with the error. If the chat_append call is the last expression in a Shiny observer, shinychat will log the error message and show a message that the error occurred in the chat UI.

Examples

```
library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Dumbest chatbot in the world: ignores user input and chooses
# a random, vague response.
fake_chatbot <- async_generator(function(input) {
  responses <- c(
    "What does that suggest to you?",
    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  await(async_sleep(1))
  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(
  chat_ui("chat", fill = TRUE)
)

server <- function(input, output, session) {
  observeEvent(input$chat_user_input, {
    response <- fake_chatbot(input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

---

chat_append_message *Low-level function to append a message to a chat control*

Description

For advanced users who want to control the message chunking behavior. Most users should use `chat_append()` instead.

Usage

```
chat_append_message(
  id,
  msg,
  chunk = TRUE,
  operation = c("append", "replace"),
  icon = NULL,
  session = getDefaultReactiveDomain()
)
```

Arguments

`id`	The ID of the chat element
`msg`	The message to append. Should be a named list with `role` and `content` fields. The `role` field should be either "user" or "assistant". The `content` field should be a string containing the message content, in Markdown format.
`chunk`	Whether `msg` is just a chunk of a message, and if so, what type. If `FALSE`, then `msg` is a complete message. If `"start"`, then `msg` is the first chunk of a multi-chunk message. If `"end"`, then `msg` is the last chunk of a multi-chunk message. If `TRUE`, then `msg` is an intermediate chunk of a multi-chunk message. Default is `FALSE`.
`operation`	The operation to perform on the message. If `"append"`, then the new content is appended to the existing message content. If `"replace"`, then the existing message content is replaced by the new content. Ignored if `chunk` is `FALSE`.
`icon`	An optional icon to display next to the message, currently only used for assistant messages. The icon can be any HTML element (e.g., `htmltools::img()` tag) or a string of HTML.
`session`	The Shiny session object

Value

Returns nothing (`invisible(NULL)`).

Examples

```
library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Dumbest chatbot in the world: ignores user input and chooses
# a random, vague response.
fake_chatbot <- async_generator(function(id, input) {
```

```
  responses <- c(
    "What does that suggest to you?",
    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  # Use low-level chat_append_message() to temporarily set a progress message
  chat_append_message(id, list(role = "assistant", content = "_Thinking..._ "))
  await(async_sleep(1))
  # Clear the progress message
  chat_append_message(id, list(role = "assistant", content = ""), operation = "replace")

  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(
  chat_ui("chat", fill = TRUE)
)

server <- function(input, output, session) {
  observeEvent(input$chat_user_input, {
    response <- fake_chatbot("chat", input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

`chat_clear` *Clear all messages from a chat control*

Description

Clear all messages from a chat control

Usage

```
chat_clear(id, session = getDefaultReactiveDomain())
```

Arguments

`id`	The ID of the chat element
`session`	The Shiny session object

Examples

```
library(shiny)
library(bslib)

ui <- page_fillable(
  chat_ui("chat", fill = TRUE),
  actionButton("clear", "Clear chat")
)

server <- function(input, output, session) {
  observeEvent(input$clear, {
    chat_clear("chat")
  })

  observeEvent(input$chat_user_input, {
    response <- paste0("You said: ", input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

---

chat_restore *Add Shiny bookmarking for shinychat*

Description

Adds Shiny bookmarking hooks to save and restore the **ellmer** chat `client`. Also restores chat messages from the history in the `client`.

If either `bookmark_on_input` or `bookmark_on_response` is `TRUE`, the Shiny App's bookmark will be automatically updated without showing a modal to the user.

Note: Only the `client`'s chat state is saved/restored in the bookmark. If the `client`'s state doesn't properly capture the chat's UI (i.e., a transformation is applied in-between receiving and displaying the message), then you may need to implement your own `session$onRestore()` (and possibly `session$onBookmark`) handler to restore any additional state.

To avoid restoring chat history from the `client`, you can ensure that the history is empty by calling `client$set_turns(list())` before passing the client to `chat_restore()`.

Usage

```
chat_restore(
  id,
  client,
  ...,
  bookmark_on_input = TRUE,
```

```
  bookmark_on_response = TRUE,
  session = getDefaultReactiveDomain()
)
```

Arguments

id	The ID of the chat element
client	The **ellmer** LLM chat client.
...	Used for future parameter expansion.
bookmark_on_input	A logical value determines if the bookmark should be updated when the user submits a message. Default is TRUE.
bookmark_on_response	A logical value determines if the bookmark should be updated when the response stream completes. Default is TRUE.
session	The Shiny session object

Value

Returns nothing (invisible(NULL)).

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- function(request) {
  page_fillable(
    chat_ui("chat", fill = TRUE)
  )
}

server <- function(input, output, session) {
  chat_client <- ellmer::chat_ollama(
    system_prompt = "Important: Always respond in a limerick",
    model = "qwen2.5-coder:1.5b",
    echo = TRUE
  )
  # Update bookmark to chat on user submission and completed response
  chat_restore("chat", chat_client)

  observeEvent(input$chat_user_input, {
    stream <- chat_client$stream_async(input$chat_user_input)
    chat_append("chat", stream)
  })
}

# Enable bookmarking!
shinyApp(ui, server, enableBookmarking = "server")
```

chat_ui *Create a chat UI element*

Description

Inserts a chat UI element into a Shiny UI, which includes a scrollable section for displaying chat messages, and an input field for the user to enter new messages.

To respond to user input, listen for `input$ID_user_input` (for example, if `id="my_chat"`, user input will be at `input$my_chat_user_input`), and use `chat_append()` to append messages to the chat.

Usage

```
chat_ui(
  id,
  ...,
  messages = NULL,
  placeholder = "Enter a message...",
  width = "min(680px, 100%)",
  height = "auto",
  fill = TRUE,
  icon_assistant = NULL
)
```

Arguments

`id`	The ID of the chat element
`...`	Extra HTML attributes to include on the chat element

`messages`	A list of messages to prepopulate the chat with. Each message can be one of the following:
	• A string, which is interpreted as markdown and rendered to HTML on the client.
	  – To prevent interpreting as markdown, mark the string as `htmltools::HTML()`.
	• A UI element.
	  – This includes `htmltools::tagList()`, which take UI elements (including strings) as children. In this case, strings are still interpreted as markdown as long as they're not inside HTML.
	• A named list of `content` and `role`. The `content` can contain content as described above, and the `role` can be "assistant" or "user".

`placeholder`	The placeholder text for the chat's user input field
`width`	The CSS width of the chat element
`height`	The CSS height of the chat element
`fill`	Whether the chat element should try to vertically fill its container, if the container is fillable

icon_assistant The icon to use for the assistant chat messages. Can be HTML or a tag in the form of `htmltools::HTML()` or `htmltools::tags()`. If None, a default robot icon is used.

Value

A Shiny tag object, suitable for inclusion in a Shiny UI

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  chat_ui("chat", fill = TRUE)
)

server <- function(input, output, session) {
  observeEvent(input$chat_user_input, {
    # In a real app, this would call out to a chat client or API,
    # perhaps using the 'ellmer' package.
    response <- paste0(
      "You said:\n\n",
      "<blockquote>",
      htmltools::htmlEscape(input$chat_user_input),
      "</blockquote>"
    )
    chat_append("chat", response)
    chat_append("chat", stream)
  })
}

shinyApp(ui, server)
```

---

contents_shinychat *Format ellmer content for shinychat*

Description

Format ellmer content for shinychat

Usage

```
contents_shinychat(content)
```

Arguments

content An `ellmer::Content` object.

Value

Returns text, HTML, or web component tags formatted for use in `chat_ui()`.

Extending `contents_shinychat()`

You can extend `contents_shinychat()` to handle custom content types in your application. `contents_shinychat()` is an S7 generic. If you haven't worked with S7 before, you can learn more about S7 classes, generics and methods in the S7 documentation.

We'll work through a short example creating a custom display for the results of a tool that gets local weather forecasts. We first need to create a custom class that extends ellmer::ContentToolResult.

```
library(ellmer)

WeatherToolResult <- S7::new_class(
  "WeatherToolResult",
  parent = ContentToolResult,
  properties = list(
    location_name = S7::class_character
  )
)
```

Next, we'll create a simple `ellmer::tool()` that gets the weather forecast for a location and returns our custom `WeatherToolResult` class. The custom class works just like a regular `ContentToolResult`, but it has an additional `location_name` property.

```
get_weather_forecast <- tool(
  function(lat, lon, location_name) {
    WeatherToolResult(
      weathR::point_tomorrow(lat, lon, short = FALSE),
      location_name = location_name
    )
  },
  name = "get_weather_forecast",
  description = "Get the weather forecast for a location.",
  arguments = list(
    lat = type_number("Latitude"),
    lon = type_number("Longitude"),
    location_name = type_string("Name of the location for display to the user")
  )
)
```

Finally, we can extend `contents_shinychat()` to render our custom content class for display in the chat interface. The basic process is to define a `contents_shinychat()` external generic and then implement a method for your custom class.

```
contents_shinychat <- S7::new_external_generic(
  package = "shinychat",
```

```
  name = "contents_shinychat",
  dispatch_args = "contents"
)

S7::method(contents_shinychat, WeatherToolResult) <- function(content) {
  # Your custom rendering logic here
}
```

You can use this pattern to completely customize how the content is displayed inside shinychat by returning HTML objects directly from this method.

You can also use this pattern to build upon the default shinychat display for tool requests and results. By using `S7::super()`, you can create the object shinychat uses for tool results (or tool requests), and then modify it to suit your needs.

```
S7::method(contents_shinychat, WeatherToolResult) <- function(content) {
  # Call the super method for ContentToolResult to get shinychat's defaults
  res <- contents_shinychat(S7::super(content, ContentToolResult))

  # Then update the result object with more specific content
  # In this case, we render the tool result dataframe as a {gt} table...
  res$value <- gt::as_raw_html(gt::gt(content@value))
  res$value_type <- "html"
  # ...and update the tool result title to include the location name
  res$title <- paste("Weather Forecast for", content@location_name)

  res
}
```

Note that you do **not** need to create a new class or extend `contents_shinychat()` to customize the tool display. Rather, you can use the strategies discussed in the Tool Calling UI article to customize the tool request and result display by providing a `display` list in the `extra` argument of the tool result.

`markdown_stream` *Stream markdown content*

Description

Streams markdown content into a `output_markdown_stream()` UI element. A markdown stream can be useful for displaying generative AI responses (outside of a chat interface), streaming logs, or other use cases where chunks of content are generated over time.

Usage

```
markdown_stream(
  id,
  content_stream,
  operation = c("replace", "append"),
  session = getDefaultReactiveDomain()
)
```

Arguments

id	The ID of the markdown stream to stream content to.
content_stream	A string generator (e.g., coro::generator() or coro::async_generator()), a string promise (e.g., promises::promise()), or a string promise generator.
operation	The operation to perform on the markdown stream. The default, "replace", will replace the current content with the new content stream. The other option, "append", will append the new content stream to the existing content.
session	The Shiny session object.

Examples

```
library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Define a generator that yields a random response
# (imagine this is a more sophisticated AI generator)
random_response_generator <- async_generator(function() {
  responses <- c(
    "What does that suggest to you?",
    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  await(async_sleep(1))
  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(
  actionButton("generate", "Generate response"),
  output_markdown_stream("stream")
)
```

```
server <- function(input, output, session) {
  observeEvent(input$generate, {
    markdown_stream("stream", random_response_generator())
  })
}

shinyApp(ui, server)
```

---

output_markdown_stream *Create a UI element for a markdown stream.*

Description

Creates a UI element for a `markdown_stream()`. A markdown stream can be useful for displaying generative AI responses (outside of a chat interface), streaming logs, or other use cases where chunks of content are generated over time.

Usage

```
output_markdown_stream(
  id,
  ...,
  content = "",
  content_type = "markdown",
  auto_scroll = TRUE,
  width = "min(680px, 100%)",
  height = "auto"
)
```

Arguments

`id`	A unique identifier for this markdown stream.
`...`	Extra HTML attributes to include on the chat element
`content`	A string of content to display before any streaming occurs. When `content_type` is Markdown or HTML, it may also be UI element(s) such as input and output bindings.
`content_type`	The content type. Default is `"markdown"` (specifically, CommonMark). Supported content types include: * `"markdown"`: markdown text, specifically CommonMark * `"html"`: for rendering HTML content. * `"text"`: for plain text. * `"semi-markdown"`: for rendering markdown, but with HTML tags escaped.
`auto_scroll`	Whether to automatically scroll to the bottom of a scrollable container when new content is added. Default is True.
`width`	The width of the UI element.
`height`	The height of the UI element.

Value

A shiny tag object.

See Also

`markdown_stream()`

`update_chat_user_input` *Update the user input of a chat control*

Description

Update the user input of a chat control

Usage

```
update_chat_user_input(
  id,
  ...,
  value = NULL,
  placeholder = NULL,
  submit = FALSE,
  focus = FALSE,
  session = getDefaultReactiveDomain()
)
```

Arguments

`id`	The ID of the chat element
`...`	Currently unused, but reserved for future use.
`value`	The value to set the user input to. If `NULL`, the input will not be updated.
`placeholder`	The placeholder text for the user input
`submit`	Whether to automatically submit the text for the user. Requires `value`.
`focus`	Whether to move focus to the input element. Requires `value`.
`session`	The Shiny session object

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  chat_ui("chat"),
  layout_columns(
```

```
    fill = FALSE,
    actionButton("update_placeholder", "Update placeholder"),
    actionButton("update_value", "Update user input")
  )
)

server <- function(input, output, session) {
  observeEvent(input$update_placeholder, {
    update_chat_user_input("chat", placeholder = "New placeholder text")
  })

  observeEvent(input$update_value, {
    update_chat_user_input("chat", value = "New user input", focus = TRUE)
  })

  observeEvent(input$chat_user_input, {
    response <- paste0("You said: ", input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

Index