Package: mini007 (via r-universe)

October 27, 2026

Type Package

Title Lightweight Framework for Orchestrating Multi-Agent Large Language Models

Version 0.5.0.9000

Description Provides tools for creating agents with persistent state using R6 classes <https://cran.r-project.org/package=R6> and the 'ellmer' package <https://cran.r-project.org/package=ellmer>. Tracks prompts, messages, and agent metadata for reproducible, multi-turn large language model sessions.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

URL https://github.com/feddelegrand7/mini007

BugReports https://github.com/feddelegrand7/mini007/issues

Imports checkmate (>= 2.3.1), cli (>= 3.6.5), coro (>= 1.0.0), DiagrammeR (>= 1.0.11), ellmer (>= 0.4.0), glue (>= 1.8.0), mirai (>= 2.4.0), R6 (>= 2.6.1), rlang (>= 1.1.6), uuid (>= 1.2.0)

RoxygenNote 7.3.2

Config/pak/sysreqs cmake libglpk-dev make libicu-dev libuv1-dev libxml2-dev libssl-dev libx11-dev

Repository https://feddelegrand7.r-universe.dev

Date/Publication 2026-10-27 08:12:39 UTC

RemoteUrl https://github.com/feddelegrand7/mini007

RemoteRef HEAD

RemoteSha 331c61acddeb4cf853d1b3b9967aeaced9a184e2

Contents

Agent 2
is_pending 22
LeadAgent 23
mini007_event 45
mini007_pending 47
Workflow 47
WorkflowAgent 52

Index **54**

`Agent` *Agent: A General-Purpose LLM Agent*

Description

The ‘Agent‘ class defines a modular LLM-based agent capable of responding to prompts using a defined role/instruction. It wraps an OpenAI-compatible chat model via the [‘ellmer‘](https://github.com/llrs/ellmer) package.

Each agent maintains its own message history and unique identity.

Public fields

`name` The agent’s name.

`instruction` The agent’s role/system prompt.

`llm_object` The underlying ‘ellmer::chat_openai‘ object.

`agent_id` A UUID uniquely identifying the agent.

`model_provider` The name of the entity providing the model (eg. OpenAI)

`model_name` The name of the model to be used (eg. gpt-4.1-mini)

`broadcast_history` A list of all past broadcast interactions.

`budget` A budget in $ that the agent should not exceed.

`budget_policy` A list controlling budget behavior: on_exceed and warn_at.

`budget_warned` Internal flag indicating whether warn_at notice was emitted.

`cost` The current cost of the agent

`tools` A list of registered tools available to the agent

Active bindings

`messages` Public active binding for the conversation history. Assignment is validated automatically.

Methods

Public methods:

- Agent$new()
- Agent$invoke()
- Agent$generate_execute_r_code()
- Agent$set_budget()
- Agent$set_budget_policy()
- Agent$keep_last_n_messages()
- Agent$add_message()
- Agent$share_context_with()
- Agent$clear_and_summarise_messages()
- Agent$update_instruction()
- Agent$get_usage_stats()
- Agent$reset_conversation_history()
- Agent$export_messages_history()
- Agent$load_messages_history()
- Agent$validate_response()
- Agent$register_tools()
- Agent$list_tools()
- Agent$remove_tools()
- Agent$clear_tools()
- Agent$generate_and_register_tool()
- Agent$clone_agent()
- Agent$clone()

Method `new()`**:** Initializes a new Agent with a specific role/instruction.

Usage:

```
Agent$new(name, instruction, llm_object, budget = NA)
```

Arguments:

`name` A short identifier for the agent (e.g. '"translator"').

`instruction` The system prompt that defines the agent's role.

`llm_object` The LLM object generate by ellmer (eg. output of ellmer::chat_openai)

`budget` Numerical value denoting the amount to set for the budget in US$ to a specific agent, if the budget is reached, an error will be thrown.

Examples:

```
# An API KEY is required in order to invoke the Agent
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
```

```
polar_bear_researcher <- Agent$new(
  name = "POLAR BEAR RESEARCHER",
  instruction = paste0(
  "You are an expert in polar bears, ",
 "you task is to collect information about polar bears. Answer in 1 sentence max."
  ),
  llm_object = openai_4_1_mini
)
```

Method `invoke()`**:** Sends a user prompt to the agent and returns the assistant's response.

Set `stream = TRUE` to receive the response incrementally: token deltas, tool calls and tool results are delivered to 'on_event' as 'mini007_event' objects while the LLM is still generating (see `mini007_event`). Without 'on_event', deltas are printed to the console and the response is returned invisibly. The returned text is the same in both modes.

Usage:

```
Agent$invoke(prompt, stream = FALSE, on_event = NULL)
```

Arguments:

`prompt` A character string prompt for the agent to respond to.

`stream` '[logical(1)]' Whether to stream the response. Defaults to 'FALSE'.

`on_event` '[function | NULL]' Callback receiving each 'mini007_event'. Supplying it implies `stream = TRUE`. Raising an error inside it cancels the run.

Returns: The LLM-generated response as a character string.

Examples:

```
\dontrun{
# An API KEY is required in order to invoke the Agent
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
)
agent <- Agent$new(
 name = "translator",
 instruction = "You are an Algerian citizen",
 llm_object = openai_4_1_mini
)
agent$invoke("Continue this sentence: 1 2 3 viva")

# Stream tokens to the console as they are generated
agent$invoke("Tell me a short story", stream = TRUE)
}
```

Method `generate_execute_r_code()`**:** Generate R code from natural language descriptions and optionally validate/execute it

Usage:

```
Agent$generate_execute_r_code(
  code_description,
  validate = FALSE,
  execute = FALSE,
  interactive = TRUE,
  env = globalenv()
)
```

Arguments:

`code_description` Character string describing the R code to generate

`validate` Logical indicating whether to validate the generated code syntax

`execute` Logical indicating whether to execute the generated code (use with caution)

`interactive` Logical; if TRUE, ask for user confirmation before executing generated code

`env` Environment in which to execute the code if execute = TRUE. Default to `globalenv`

Returns: A list containing the generated code and validation/execution results

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
r_assistant <- Agent$new(
  name = "R Code Assistant",
  instruction = paste("You are an expert R programmer",
  llm_object = openai_4_1_mini
)
# Generate code for data manipulation
result <- r_assistant$generate_execute_r_code(
  code_description = "Calculate the summary of the mtcars dataframe",
  validate = TRUE,
  execute = TRUE,
  interactive = TRUE
)
print(result)
}
```

Method `set_budget()`**:** Set a budget to a specific agent, if the budget is reached, an error will be thrown

Usage:

```
Agent$set_budget(amount_in_usd)
```

Arguments:

`amount_in_usd` Numerical value denoting the amount to set for the budget,

Examples:

```
\dontrun{
# An API KEY is required in order to invoke the Agent
```

```
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
)
agent <- Agent$new(
 name = "translator",
 instruction = "You are an Algerian citizen",
 llm_object = openai_4_1_mini
)
agent$set_budget(amount_in_usd = 10.5) # this is equivalent to 10.5$
}
```

Method `set_budget_policy()`**:** Configure how the agent behaves as it approaches or exceeds its budget. Use ‘warn_at‘ (0-1) to emit a one-time warning when spending reaches the specified fraction of the budget. When the budget is exceeded, ‘on_exceed‘ controls behavior: abort, warn and proceed, or ask interactively.

Usage:

```
Agent$set_budget_policy(on_exceed = "abort", warn_at = 0.8)
```

Arguments:

`on_exceed` One of "abort", "warn", or "ask".

`warn_at` Numeric in (0,1); fraction of budget to warn at. Default 0.8.

Examples:

```
\dontrun{
agent$set_budget(5)
agent$set_budget_policy(on_exceed = "ask", warn_at = 0.9)
}
```

Method `keep_last_n_messages()`**:** Keep only the most recent ‘n‘ messages, discarding older ones while keeping the system prompt.

Usage:

```
Agent$keep_last_n_messages(n = 2)
```

Arguments:

`n` Number of most recent messages to keep.

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital finder",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
```

```
)
agent$invoke("What is the capital of Algeria")
agent$invoke("What is the capital of Germany")
agent$invoke("What is the capital of Italy")
agent$keep_last_n_messages(n = 2)
}
```

Method `add_message()`**:** Add a pre-formatted message to the conversation history

Usage:

```
Agent$add_message(role, content)
```

Arguments:

`role` The role of the message ("user", "assistant", or "system")

`content` The content of the message

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "AI assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$add_message("user", "Hello, how are you?")
agent$add_message("assistant", "I'm doing well, thank you!")
}
```

Method `share_context_with()`**:** Share the most recent conversation messages with another agent.

The last `n` non-system messages from `self` are appended to the history of an other `agent`. This is useful when one agent needs to handoff context or continue a thread in a separate agent.

Usage:

```
Agent$share_context_with(agent, n = 5)
```

Arguments:

`agent` An object of class `Agent` that will receive the messages.

`n` Number of recent messages to transfer (excluding the system prompt). Defaults to `5`.

Examples:

```
\dontrun{
llm_object <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
```

```
a1 <- Agent$new("a1", "instr", llm_object)
a2 <- Agent$new("a2", "instr", llm_object)
a1$invoke("Hello")
a1$invoke("How are you?")
a1$share_context_with(a2, n = 2)
a2$messages
}
```

Method `clear_and_summarise_messages()`**:** Summarises the agent's conversation history into a concise form and appends it to the system prompt. Unlike 'update_instruction()', this method does not override the existing instruction but augments it with a summary for future context.

After creating the summary, the method clears the conversation history and retains only the updated system prompt. This ensures that subsequent interactions start fresh but with the summary preserved as context.

Usage:

```
Agent$clear_and_summarise_messages()
```

Examples:

```
\dontrun{
  # Requires an OpenAI-compatible LLM from `ellmer`
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )

  agent <- Agent$new(
    name = "summariser",
    instruction = "You are a summarising assistant",
    llm_object = openai_4_1_mini
  )

  agent$invoke("The quick brown fox jumps over the lazy dog.")
  agent$invoke("This is another example sentence.")

  # Summarises and resets history
  agent$summarise_messages()

  # Now only the system prompt (with summary) remains
  agent$messages
}
```

Method `update_instruction()`**:** Update the system prompt/instruction

Usage:

```
Agent$update_instruction(new_instruction)
```

Arguments:

`new_instruction` New instruction to use. Not that the new instruction will override the old one

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$update_instruction("You are a concise assistant.")
}
```

Method `get_usage_stats()`**:** Get the current token count and estimated cost of the conversation

Usage:

```
Agent$get_usage_stats()
```

Returns: A list with token counts and cost information

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$set_budget(1)
agent$invoke("What is the capital of Algeria?")
stats <- agent$get_usage_stats()
stats
}
```

Method `reset_conversation_history()`**:** Reset the agent's conversation history while keeping the system instruction

Usage:

```
Agent$reset_conversation_history()
```

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
```

```
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "AI assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$invoke("Hello, how are you?")
agent$invoke("Tell me about machine learning")
agent$reset_conversation_history()  # Clears all messages except system prompt
}
```

Method `export_messages_history()`**:** Saves the agent's current conversation history as a JSON file on disk.

Usage:

```
Agent$export_messages_history(
  file_path = paste0(getwd(), "/", paste0(self$name, "_messages.json"))
)
```

Arguments:

`file_path` Character string specifying the file path where the JSON file should be saved. Defaults to a file named ‘"<agent_name>_messages.json"‘ in the current working directory.

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital_finder",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$invoke("What is the capital of Algeria")
agent$invoke("What is the capital of Italy")
agent$export_messages_history()
}
```

Method `load_messages_history()`**:** Load an agent's conversation history as a JSON file from disk.

Usage:

```
Agent$load_messages_history(
  file_path = paste0(getwd(), "/", paste0(self$name, "_messages.json"))
)
```

Arguments:

`file_path` Character string specifying the file path where the JSON file is stored. Defaults to a file named ‘"<agent_name>_messages.json"‘ in the current working directory.

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital_finder",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
# use the export_messages_history to save the interaction first
agent$load_messages_history("path/to/messages.json")
agent$messages
agent$llm_object
}
```

Method `validate_response()`**:** Validates an agent's response against custom criteria using LLM-based validation. This method uses the agent's LLM to evaluate whether a response meets specified validation criteria (e.g., accuracy, completeness, tone, format).

Usage:

```
Agent$validate_response(
  prompt,
  response,
  validation_criteria,
  validation_score = 0.8
)
```

Arguments:

`prompt` The prompt used to generate the response.

`response` The response text to validate.

`validation_criteria` The criteria for validation. (e.g., "The response should be accurate and complete", "The response must be under 100 words").

`validation_score` A numeric from 0 to 1 denoting the score to consider for the evaluation. During the evaluation, the agent will provide a score from 0 to 1, a provided score greater or equal to the ‘validation_score‘ will result in a ‘valid‘ response. Defaults to 0.8.

Returns: list object

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
```

```
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "fact_checker",
  instruction = "You are a factual assistant.",
  llm_object = openai_4_1_mini
)
prompt <- "What is the capital of Algeria?"
response <- agent$invoke(prompt)
validation <- agent$validate_response(
  response = response,
  prompt = prompt,
  validation_criteria = "The response must be accurate and mention Algiers",
  validation_score = 0.8
)
print(validation)
}
```

Method `register_tools()`**:** Register one or more tools with the agent

Usage:

```
Agent$register_tools(tools)
```

Arguments:

`tools` A list of ellmer tool objects or a single tool object

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "file_assistant",
  instruction = "You are a file management assistant.",
  llm_object = openai_4_1_mini
)
# Register predefined tools
agent$register_tools(list(change_directory_tool(), list_files_tool()))
}
```

Method `list_tools()`**:** List all registered tools

Usage:

```
Agent$list_tools()
```

Returns: Character vector of tool names

Examples:

```
\dontrun{
agent$list_tools()
}
```

Method `remove_tools()`**:** Remove tools from the agent

Usage:

```
Agent$remove_tools(tool_names)
```

Arguments:

`tool_names` Character vector of tool names to remove

Examples:

```
\dontrun{
agent$remove_tools(c("change_directory", "list_files"))
}
```

Method `clear_tools()`**:** Clear all registered tools

Usage:

```
Agent$clear_tools()
```

Examples:

```
\dontrun{
agent$clear_tools()
}
```

Method `generate_and_register_tool()`**:** Generate and register a tool from a natural language description. This method uses the agent's LLM to create both the R function and the ellmer tool definition based on your description, then automatically registers it.

Usage:

```
Agent$generate_and_register_tool(description)
```

Arguments:

`description` Character string describing what the tool should do

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "file_assistant",
  instruction = "You are a helpful assistant.",
  llm_object = openai_4_1_mini
)
# Generate and register a tool from description
agent$generate_and_register_tool(
  description = "Create a tool that saves text content to a file on disk"
)
```

```
# Now the tool is available to use
agent$list_tools()
}
```

Method `clone_agent()`**:** Create a copy of the agent with the same instruction and configuration but a new unique ID. Useful for creating multiple instances of the same agent type.

Usage:

```
Agent$clone_agent(new_name = NULL)
```

Arguments:

`new_name` Optional character string to assign a new name to the cloned agent. If NULL, the cloned agent retains the original name.

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "translator",
  instruction = "You are a translator.",
  llm_object = openai_4_1_mini
)
# Clone with same name
agent_copy <- agent$clone_agent()
}
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

[load_messages_history()] for reloading a saved message history.

[export_messages_history()] for exporting the messages object to json.

Examples

```
## ------------------------------------------------
## Method `Agent$new`
## ------------------------------------------------

  # An API KEY is required in order to invoke the Agent
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
```

```
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )

  polar_bear_researcher <- Agent$new(
    name = "POLAR BEAR RESEARCHER",
    instruction = paste0(
    "You are an expert in polar bears, ",
    "you task is to collect information about polar bears. Answer in 1 sentence max."
    ),
    llm_object = openai_4_1_mini
  )


## ------------------------------------------------
## Method `Agent$invoke`
## ------------------------------------------------

## Not run:
# An API KEY is required in order to invoke the Agent
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
)
agent <- Agent$new(
 name = "translator",
 instruction = "You are an Algerian citizen",
 llm_object = openai_4_1_mini
)
agent$invoke("Continue this sentence: 1 2 3 viva")

# Stream tokens to the console as they are generated
agent$invoke("Tell me a short story", stream = TRUE)

## End(Not run)

## ------------------------------------------------
## Method `Agent$generate_execute_r_code`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
r_assistant <- Agent$new(
  name = "R Code Assistant",
  instruction = paste("You are an expert R programmer",
  llm_object = openai_4_1_mini
)
# Generate code for data manipulation
```

```
result <- r_assistant$generate_execute_r_code(
  code_description = "Calculate the summary of the mtcars dataframe",
  validate = TRUE,
  execute = TRUE,
  interactive = TRUE
)
print(result)

## End(Not run)

## ------------------------------------------------
## Method `Agent$set_budget`
## ------------------------------------------------

## Not run:
# An API KEY is required in order to invoke the Agent
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
)
agent <- Agent$new(
 name = "translator",
 instruction = "You are an Algerian citizen",
 llm_object = openai_4_1_mini
)
agent$set_budget(amount_in_usd = 10.5) # this is equivalent to 10.5$

## End(Not run)

## ------------------------------------------------
## Method `Agent$set_budget_policy`
## ------------------------------------------------

## Not run:
agent$set_budget(5)
agent$set_budget_policy(on_exceed = "ask", warn_at = 0.9)

## End(Not run)

## ------------------------------------------------
## Method `Agent$keep_last_n_messages`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital finder",
  instruction = "You are an assistant.",
```

```
  llm_object = openai_4_1_mini
)
agent$invoke("What is the capital of Algeria")
agent$invoke("What is the capital of Germany")
agent$invoke("What is the capital of Italy")
agent$keep_last_n_messages(n = 2)

## End(Not run)

## ------------------------------------------------
## Method `Agent$add_message`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "AI assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$add_message("user", "Hello, how are you?")
agent$add_message("assistant", "I'm doing well, thank you!")

## End(Not run)

## ------------------------------------------------
## Method `Agent$share_context_with`
## ------------------------------------------------

## Not run:
llm_object <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
a1 <- Agent$new("a1", "instr", llm_object)
a2 <- Agent$new("a2", "instr", llm_object)
a1$invoke("Hello")
a1$invoke("How are you?")
a1$share_context_with(a2, n = 2)
a2$messages

## End(Not run)

## ------------------------------------------------
## Method `Agent$clear_and_summarise_messages`
## ------------------------------------------------

## Not run:
```

```
  # Requires an OpenAI-compatible LLM from `ellmer`
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )

  agent <- Agent$new(
    name = "summariser",
    instruction = "You are a summarising assistant",
    llm_object = openai_4_1_mini
  )

  agent$invoke("The quick brown fox jumps over the lazy dog.")
  agent$invoke("This is another example sentence.")

  # Summarises and resets history
  agent$summarise_messages()

  # Now only the system prompt (with summary) remains
  agent$messages

## End(Not run)


## ------------------------------------------------
## Method `Agent$update_instruction`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$update_instruction("You are a concise assistant.")

## End(Not run)

## ------------------------------------------------
## Method `Agent$get_usage_stats`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
```

```
)
agent <- Agent$new(
  name = "assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$set_budget(1)
agent$invoke("What is the capital of Algeria?")
stats <- agent$get_usage_stats()
stats

## End(Not run)

## ------------------------------------------------
## Method `Agent$reset_conversation_history`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "AI assistant",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$invoke("Hello, how are you?")
agent$invoke("Tell me about machine learning")
agent$reset_conversation_history()  # Clears all messages except system prompt

## End(Not run)

## ------------------------------------------------
## Method `Agent$export_messages_history`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital_finder",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
agent$invoke("What is the capital of Algeria")
agent$invoke("What is the capital of Italy")
agent$export_messages_history()
```

```
## End(Not run)


## ------------------------------------------------
## Method `Agent$load_messages_history`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "capital_finder",
  instruction = "You are an assistant.",
  llm_object = openai_4_1_mini
)
# use the export_messages_history to save the interaction first
agent$load_messages_history("path/to/messages.json")
agent$messages
agent$llm_object

## End(Not run)


## ------------------------------------------------
## Method `Agent$validate_response`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "fact_checker",
  instruction = "You are a factual assistant.",
  llm_object = openai_4_1_mini
)
prompt <- "What is the capital of Algeria?"
response <- agent$invoke(prompt)
validation <- agent$validate_response(
  response = response,
  prompt = prompt,
  validation_criteria = "The response must be accurate and mention Algiers",
  validation_score = 0.8
)
print(validation)

## End(Not run)
```

```
## ------------------------------------------------
## Method `Agent$register_tools`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "file_assistant",
  instruction = "You are a file management assistant.",
  llm_object = openai_4_1_mini
)
# Register predefined tools
agent$register_tools(list(change_directory_tool(), list_files_tool()))

## End(Not run)

## ------------------------------------------------
## Method `Agent$list_tools`
## ------------------------------------------------

## Not run:
agent$list_tools()

## End(Not run)

## ------------------------------------------------
## Method `Agent$remove_tools`
## ------------------------------------------------

## Not run:
agent$remove_tools(c("change_directory", "list_files"))

## End(Not run)

## ------------------------------------------------
## Method `Agent$clear_tools`
## ------------------------------------------------

## Not run:
agent$clear_tools()

## End(Not run)

## ------------------------------------------------
## Method `Agent$generate_and_register_tool`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
```

```
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "file_assistant",
  instruction = "You are a helpful assistant.",
  llm_object = openai_4_1_mini
)
# Generate and register a tool from description
agent$generate_and_register_tool(
  description = "Create a tool that saves text content to a file on disk"
)
# Now the tool is available to use
agent$list_tools()

## End(Not run)

## ------------------------------------------------
## Method `Agent$clone_agent`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
agent <- Agent$new(
  name = "translator",
  instruction = "You are a translator.",
  llm_object = openai_4_1_mini
)
# Clone with same name
agent_copy <- agent$clone_agent()

## End(Not run)
```

`is_pending` *Check whether a value is a pending HITL request*

Description

Returns 'TRUE' if 'x' is a 'mini007_pending' object, as returned by `Workflow$run()`, `Workflow$resume()`, `LeadAgent$invoke()`, or `LeadAgent$resume()` when execution pauses for human input.

Usage

```
is_pending(x)
```

Arguments

x	An object to test.

Value

'[logical(1)]'

Examples

```
wf <- Workflow$new("W")
wf$add_station("s1", function(x) x)
wf$set_hitl(1L, mode = "pause")
result <- wf$run("hello")
is_pending(result)
```

LeadAgent *LeadAgent: A Multi-Agent Orchestration Coordinator*

Description

'LeadAgent' extends 'Agent' to coordinate a group of specialized agents. It decomposes complex prompts into subtasks using LLMs and assigns each subtask to the most suitable registered agent. The lead agent handles response chaining, where each agent can consider prior results.

Details

This class builds intelligent multi-agent workflows by delegating sub-tasks using 'delegate_prompt()', executing them with 'invoke()', and storing the results in the 'agents_interaction' list.

Super class

`mini007::Agent` -> `LeadAgent`

Public fields

`agents` A named list of registered sub-agents (by UUID).

`agents_interaction` A list of delegated task history with agent IDs, prompts, and responses.

`plan` A list containing the most recently generated task plan.

`hitl_steps` The steps where the workflow should be stopped in order to allow for a human interaction

`hitl_mode` Either '"console"' (blocking `readline()` prompt, the default) or '"pause"' (non-blocking: `$invoke()` returns a 'mini007_pending' object instead of blocking). Set via `$set_hitl()`.

`prompt_for_plan` The prompt used to generate the plan.

`agents_for_plan` The agents used for the plan

`dialog_history` A list storing the history of agent dialogs

`broadcast_history` A list storing the history of broadcast interactions

`n_daemons` Number of parallel daemons used for parallel broadcast. Set via `$set_daemons()`.

Methods

Public methods:

- LeadAgent$new()
- LeadAgent$clear_agents()
- LeadAgent$remove_agents()
- LeadAgent$register_agents()
- LeadAgent$visualize_plan()
- LeadAgent$invoke()
- LeadAgent$resume()
- LeadAgent$generate_plan()
- LeadAgent$set_daemons()
- LeadAgent$broadcast()
- LeadAgent$set_hitl()
- LeadAgent$judge_and_choose_best_response()
- LeadAgent$agents_dialog()
- LeadAgent$clone()

Method `new()`**:** Initializes the LeadAgent with a built-in task-decomposition prompt.

Usage:

```
LeadAgent$new(name, llm_object)
```

Arguments:

`name` A short name for the coordinator (e.g. '"lead"').

`llm_object` The LLM object generate by ellmer (eg. output of ellmer::chat_openai)

Examples:

```
  # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )
```

Method `clear_agents()`**:** Clear out the registered Agents

Usage:

```
LeadAgent$clear_agents()
```

Examples:

```
 # An API KEY is required in order to invoke the agents
 openai_4_1_mini <- ellmer::chat(
   name = "openai/gpt-4.1-mini",
   api_key = Sys.getenv("OPENAI_API_KEY"),
   echo = "none"
 )
researcher <- Agent$new(
  name = "researcher",
  instruction = paste0(
  "You are a research assistant. ",
 "Your job is to answer factual questions with detailed and accurate information. ",
  "Do not answer with more than 2 lines"
  ),
  llm_object = openai_4_1_mini
)

summarizer <- Agent$new(
  name = "summarizer",
  instruction = paste0(
  "You are an agent designed to summarise ",
  "a given text into 3 distinct bullet points."
  ),
  llm_object = openai_4_1_mini
)

translator <- Agent$new(
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)
lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

lead_agent$agents

lead_agent$clear_agents()

lead_agent$agents
```

Method `remove_agents()`**:** Remove registered agents by IDs

Usage:

```
LeadAgent$remove_agents(agent_ids)
```

Arguments:

agent_ids The Agent ID to remove from the registered Agents

Examples:

```
 # An API KEY is required in order to invoke the agents
 openai_4_1_mini <- ellmer::chat(
   name = "openai/gpt-4.1-mini",
   api_key = Sys.getenv("OPENAI_API_KEY"),
   echo = "none"
 )
researcher <- Agent$new(
  name = "researcher",
  instruction = paste0(
  "You are a research assistant. ",
 "Your job is to answer factual questions with detailed and accurate information. ",
  "Do not answer with more than 2 lines"
  ),
  llm_object = openai_4_1_mini
)

summarizer <- Agent$new(
  name = "summarizer",
 instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
  llm_object = openai_4_1_mini
)

translator <- Agent$new(
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)


lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

lead_agent$agents

# deleting the translator agent

id_translator_agent <- translator$agent_id

lead_agent$remove_agents(id_translator_agent)

lead_agent$agents
```

Method `register_agents()`**:** Register one or more agents for delegation.

Usage:

```
LeadAgent$register_agents(agents)
```

Arguments:

`agents` A vector of 'Agent' objects to register.

Examples:

```
  # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. ",
  "Your job is to answer factual questions with detailed and accurate information. ",
   "Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
  instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 lead_agent$agents
```

Method `visualize_plan()`**:** Visualizes the orchestration plan Each agent node is shown in sequence (left → right), with tooltips showing the actual prompt delegated to that agent.

Usage:

```
LeadAgent$visualize_plan()
```

Method `invoke()`**:** Executes the full prompt pipeline: decomposition → delegation → invocation.

With `stream = TRUE` (or an 'on_event' callback), progress is delivered as 'mini007_event' objects: the plan, the start/completion of every delegated task, HITL requests, the final output, and the token deltas of every delegated Agent (see `mini007_event`). The lead agent's own planning calls are not streamed.

Usage:

```
LeadAgent$invoke(
  prompt,
  force_regenerate_plan = FALSE,
  stream = FALSE,
  on_event = NULL
)
```

Arguments:

`prompt` The complex user instruction to process.

`force_regenerate_plan` If TRUE, regenerate a plan even if one exists, defaults to FALSE.

`stream` '[logical(1)]' Whether to stream progress. Without 'on_event', Agent token deltas are printed to the console.

`on_event` '[function | NULL]' Callback receiving each 'mini007_event'. Supplying it implies `stream = TRUE`. Raising an error inside it cancels the run.

Returns: The final response (from the last agent in the sequence).

Examples:

```
\dontrun{
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. ",
   "Your job is to answer factual questions with detailed ",
   "and accurate information. Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
  instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
```

```
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)

lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

lead_agent$invoke(
paste0(
 "Describe the economic situation in Algeria in 3 sentences. ",
 "Answer in German"
 )
)
}
```

Method `resume()`**:** Resume a ‘LeadAgent‘ paused at an HITL step.
Only relevant when `hitl_mode = "pause"` (see `$set_hitl()`). `$invoke()` returns a ‘mini007_pending‘ object when it reaches a paused HITL step; call `$resume()` with that object’s ‘request_id‘ and a chosen ‘action‘ to continue delegation from that point.

Usage:

```
LeadAgent$resume(
  request_id,
  action = c("continue", "edit", "abort"),
  value = NULL,
  stream = FALSE,
  on_event = NULL
)
```

Arguments:

`request_id` ‘[character(1)]‘ The ‘request_id‘ from the ‘mini007_pending‘ object returned by `$invoke()`.

`action` ‘[character(1)]‘ One of ‘"continue"‘ (use the proposed response as-is), ‘"edit"‘ (replace it with ‘value‘), or ‘"abort"‘ (raise an error, stopping delegation).

`value` ‘[character(1) | NULL]‘ Required when `action = "edit"`: the replacement response.

`stream` ‘[logical(1)]‘ Whether to stream the remaining steps (see `$invoke()`).

`on_event` ‘[function | NULL]‘ Callback receiving each ‘mini007_event‘. Supplying it implies `stream = TRUE`.

Returns: ‘[character(1)]‘ The final response, or another ‘mini007_pending‘ object if a further HITL step is reached.

Method `generate_plan()`**:** Generates a task execution plan without executing the subtasks. It returns a structured list containing the subtask, the selected agent, and metadata.

Usage:

```
LeadAgent$generate_plan(prompt)
```

Arguments:

`prompt` A complex instruction to be broken into subtasks.

Returns: A list of lists containing agent_id, agent_name, model_name, model_provider, and the assigned prompt.

Examples:

```
\dontrun{
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. Your job is to answer factual questions ",
  "with detailed and accurate information. Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
  instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 lead_agent$generate_plan(
 paste0(
  "Describe the economic situation in Algeria in 3 sentences. ",
  "Answer in German"
  )
 )
```

```
}
```

Method `set_daemons()`**:** Configure parallel execution daemons for parallel broadcast using mirai.

Usage:

```
LeadAgent$set_daemons(n = 0L)
```

Arguments:

`n` '[integerish(1)]' Number of daemon processes (default '0'). Use '0' to shut down daemons.

Returns: Invisibly returns 'self'.

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
lead_agent <- LeadAgent$new(name = "Leader", llm_object = openai_4_1_mini)
lead_agent$set_daemons(4)
}
```

Method `broadcast()`**:** Broadcasts a prompt to all registered agents and collects their responses. This does not affect the main agent orchestration logic or history.

Usage:

```
LeadAgent$broadcast(prompt, parallel = FALSE, synthesize = FALSE)
```

Arguments:

`prompt` A user prompt to send to all agents.

`parallel` '[logical(1)]' When 'TRUE', agents are invoked concurrently via mirai daemons. Requires '$set_daemons(n > 0)' to be called first. Default 'FALSE'.

`synthesize` '[logical(1)]' When 'TRUE', returns a single character string summarising each agent's response in the form "agent_name answered with ...". Default 'FALSE'.

Returns: A list of responses from all agents, or a single character string when 'synthesize = TRUE'.

Examples:

```
\dontrun{
 # An API KEY is required in order to invoke the agents
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
openai_4_1 <- ellmer::chat(
  name = "openai/gpt-4.1",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
```

```
)

openai_4_1_agent <- Agent$new(
  name = "openai_4_1_agent",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1
)

openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

openai_4_1_nano_agent <- Agent$new(
  name = "openai_4_1_nano_agent",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1_nano
  )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

lead_agent$register_agents(c(openai_4_1_agent, openai_4_1_nano_agent))
lead_agent$broadcast(
  prompt = paste0(
    "If I were Algerian, which song would I like to sing ",
    "when running under the rain? how about a flower?"
  )
  )
}
```

Method `set_hitl()`**:** Set Human In The Loop (HITL) interaction at determined steps within the workflow. With `mode = "console"` (default) a paused step blocks on `readline()`, unchanged from previous behavior. With `mode = "pause"`, `$invoke()` instead returns a 'mini007_pending' object describing the paused step; call `$resume()` with the chosen action to continue delegation. This allows HITL to work outside an interactive console.

Usage:

```
LeadAgent$set_hitl(steps, mode = c("console", "pause"))
```

Arguments:

`steps` At which steps the Human In The Loop is required?

`mode` '[character(1)]' '"console"' (default) or '"pause"'.

Returns: A list of responses from all agents.

Examples:

```
\dontrun{
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
    "You are a research assistant. ",
   "Your job is to answer factual questions with detailed and accurate information. ",
    "Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
   instruction = paste0(
   "You are agent designed to summarise a give text ",
   "into 3 distinct bullet points."
   ),
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 # setting a human in the loop in step 2
 lead_agent$set_hitl(1)

 # The execution will stop at step 2 and a human will be able
 # to either accept the answer, modify it or stop the execution of
 # the workflow

 lead_agent$invoke(
 paste0(
```

```
  "Describe the economic situation in Algeria in 3 sentences. ",
  "Answer in German"
  )
 )
}
```

Method `judge_and_choose_best_response()`**:** The Lead Agent send a prompt to its registered agents and choose the best response from the agents' responses

Usage:

```
LeadAgent$judge_and_choose_best_response(prompt)
```

Arguments:

`prompt` The prompt to send to the registered agents

Returns: A list of responses from all agents, including the chosen response

Examples:

```
\dontrun{
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
openai_4_1 <- ellmer::chat(
  name = "openai/gpt-4.1",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

stylist <- Agent$new(
  name = "stylist",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1
)

openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

stylist2 <- Agent$new(
  name = "stylist2",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1_nano
)

lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
```

```
)

lead_agent$register_agents(c(stylist, stylist2))

lead_agent$judge_and_choose_best_response("what's the best way to war a kalvin klein shirt?")

}
```

Method `agents_dialog()`**:** Facilitates a collaborative dialog between two agents to refine a response. The agents take turns building on each other's responses until they reach consensus or the maximum iterations are reached. Agents can signal consensus by starting their response with "CONSENSUS:". If max iterations is reached without consensus, the lead agent synthesizes a final response.

Usage:

```
LeadAgent$agents_dialog(prompt, agent_1_id, agent_2_id, max_iterations = 5)
```

Arguments:

`prompt` The initial task or question for the agents to discuss.

`agent_1_id` The ID of the first agent to participate in the dialog.

`agent_2_id` The ID of the second agent to participate in the dialog.

`max_iterations` Maximum number of back-and-forth exchanges (default: 5).

Returns: A list containing the final response, consensus status, and complete dialog history.

Examples:

```
\dontrun{
# An API KEY is required in order to invoke the agents
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

creative_writer <- Agent$new(
  name = "creative_writer",
  instruction = "You are a creative writer. Focus on engaging storytelling.",
  llm_object = openai_4_1_nano
)

editor <- Agent$new(
  name = "editor",
  instruction = "You are an editor. Focus on clarity and conciseness.",
  llm_object = openai_4_1_mini
)
```

```
lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(creative_writer, editor))

result <- lead_agent$agents_dialog(
  prompt = "Write a compelling opening sentence for a sci-fi novel.",
  agent_1_id = creative_writer$agent_id,
  agent_2_id = editor$agent_id,
  max_iterations = 3
)

# Access the final response
result$final_response

# View the dialog history
result$dialog_history
}
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `LeadAgent$new`
## ------------------------------------------------


 # An API KEY is required in order to invoke the agents
 openai_4_1_mini <- ellmer::chat(
   name = "openai/gpt-4.1-mini",
   api_key = Sys.getenv("OPENAI_API_KEY"),
   echo = "none"
 )

lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)



## ------------------------------------------------
```

```
## Method `LeadAgent$clear_agents`
## ------------------------------------------------

  # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. ",
   "Your job is to answer factual questions with detailed and accurate information. ",
   "Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
   instruction = paste0(
   "You are an agent designed to summarise ",
   "a given text into 3 distinct bullet points."
   ),
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )
 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 lead_agent$agents

 lead_agent$clear_agents()

 lead_agent$agents


## ------------------------------------------------
## Method `LeadAgent$remove_agents`
## ------------------------------------------------

  # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
```

```
   name = "openai/gpt-4.1-mini",
   api_key = Sys.getenv("OPENAI_API_KEY"),
   echo = "none"
 )
researcher <- Agent$new(
  name = "researcher",
  instruction = paste0(
  "You are a research assistant. ",
  "Your job is to answer factual questions with detailed and accurate information. ",
  "Do not answer with more than 2 lines"
  ),
  llm_object = openai_4_1_mini
)

summarizer <- Agent$new(
  name = "summarizer",
 instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
  llm_object = openai_4_1_mini
)

translator <- Agent$new(
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)


lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

lead_agent$agents

# deleting the translator agent

id_translator_agent <- translator$agent_id

lead_agent$remove_agents(id_translator_agent)

lead_agent$agents


## ------------------------------------------------
## Method `LeadAgent$register_agents`
## ------------------------------------------------

  # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
```

```
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. ",
   "Your job is to answer factual questions with detailed and accurate information. ",
   "Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
  instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
   llm_object = openai_4_1_mini
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 lead_agent$agents

## ------------------------------------------------
## Method `LeadAgent$invoke`
## ------------------------------------------------

## Not run:
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. ",
   "Your job is to answer factual questions with detailed ",
   "and accurate information. Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )
```

```
summarizer <- Agent$new(
  name = "summarizer",
 instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
  llm_object = openai_4_1_mini
)

translator <- Agent$new(
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)

lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

lead_agent$invoke(
paste0(
 "Describe the economic situation in Algeria in 3 sentences. ",
 "Answer in German"
 )
)

## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$generate_plan`
## ------------------------------------------------

## Not run:
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
 )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
   "You are a research assistant. Your job is to answer factual questions ",
   "with detailed and accurate information. Do not answer with more than 2 lines"
   ),
   llm_object = openai_4_1_mini
 )

 summarizer <- Agent$new(
   name = "summarizer",
  instruction = "You are agent designed to summarise a given text into 3 distinct bullet points.",
   llm_object = openai_4_1_mini
```

```
 )

 translator <- Agent$new(
   name = "translator",
   instruction = "Your role is to translate a text from English to German",
   llm_object = openai_4_1_mini
 )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

 lead_agent$register_agents(c(researcher, summarizer, translator))

 lead_agent$generate_plan(
 paste0(
  "Describe the economic situation in Algeria in 3 sentences. ",
  "Answer in German"
  )
 )

## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$set_daemons`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
lead_agent <- LeadAgent$new(name = "Leader", llm_object = openai_4_1_mini)
lead_agent$set_daemons(4)

## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$broadcast`
## ------------------------------------------------

## Not run:
 # An API KEY is required in order to invoke the agents
openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
openai_4_1 <- ellmer::chat(
  name = "openai/gpt-4.1",
  api_key = Sys.getenv("OPENAI_API_KEY"),
```

```
  echo = "none"
)

openai_4_1_agent <- Agent$new(
  name = "openai_4_1_agent",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1
)

openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

openai_4_1_nano_agent <- Agent$new(
  name = "openai_4_1_nano_agent",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1_nano
  )

 lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
 )

lead_agent$register_agents(c(openai_4_1_agent, openai_4_1_nano_agent))
lead_agent$broadcast(
  prompt = paste0(
    "If I were Algerian, which song would I like to sing ",
    "when running under the rain? how about a flower?"
  )
  )

## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$set_hitl`
## ------------------------------------------------

## Not run:
 # An API KEY is required in order to invoke the agents
  openai_4_1_mini <- ellmer::chat(
    name = "openai/gpt-4.1-mini",
    api_key = Sys.getenv("OPENAI_API_KEY"),
    echo = "none"
  )
 researcher <- Agent$new(
   name = "researcher",
   instruction = paste0(
    "You are a research assistant. ",
    "Your job is to answer factual questions with detailed and accurate information. ",
    "Do not answer with more than 2 lines"
```

```
  ),
  llm_object = openai_4_1_mini
)

summarizer <- Agent$new(
  name = "summarizer",
  instruction = paste0(
  "You are agent designed to summarise a give text ",
  "into 3 distinct bullet points."
  ),
  llm_object = openai_4_1_mini
)

translator <- Agent$new(
  name = "translator",
  instruction = "Your role is to translate a text from English to German",
  llm_object = openai_4_1_mini
)

lead_agent <- LeadAgent$new(
 name = "Leader",
 llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(researcher, summarizer, translator))

# setting a human in the loop in step 2
lead_agent$set_hitl(1)

# The execution will stop at step 2 and a human will be able
# to either accept the answer, modify it or stop the execution of
# the workflow

lead_agent$invoke(
paste0(
 "Describe the economic situation in Algeria in 3 sentences. ",
 "Answer in German"
 )
)

## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$judge_and_choose_best_response`
## ------------------------------------------------

## Not run:
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
openai_4_1 <- ellmer::chat(
```

```
  name = "openai/gpt-4.1",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

stylist <- Agent$new(
  name = "stylist",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1
)

openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

stylist2 <- Agent$new(
  name = "stylist2",
  instruction = "You are an AI assistant. Answer in 1 sentence max.",
  llm_object = openai_4_1_nano
)

lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(stylist, stylist2))

lead_agent$judge_and_choose_best_response("what's the best way to war a kalvin klein shirt?")


## End(Not run)

## ------------------------------------------------
## Method `LeadAgent$agents_dialog`
## ------------------------------------------------

## Not run:
# An API KEY is required in order to invoke the agents
openai_4_1_mini <- ellmer::chat(
  name = "openai/gpt-4.1-mini",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)
openai_4_1_nano <- ellmer::chat(
  name = "openai/gpt-4.1-nano",
  api_key = Sys.getenv("OPENAI_API_KEY"),
  echo = "none"
)

creative_writer <- Agent$new(
```

```
  name = "creative_writer",
  instruction = "You are a creative writer. Focus on engaging storytelling.",
  llm_object = openai_4_1_nano
)

editor <- Agent$new(
  name = "editor",
  instruction = "You are an editor. Focus on clarity and conciseness.",
  llm_object = openai_4_1_mini
)

lead_agent <- LeadAgent$new(
  name = "Leader",
  llm_object = openai_4_1_mini
)

lead_agent$register_agents(c(creative_writer, editor))

result <- lead_agent$agents_dialog(
  prompt = "Write a compelling opening sentence for a sci-fi novel.",
  agent_1_id = creative_writer$agent_id,
  agent_2_id = editor$agent_id,
  max_iterations = 3
)

# Access the final response
result$final_response

# View the dialog history
result$dialog_history

## End(Not run)
```

`mini007_event` *Streaming Events*

Description

'Agent$invoke()', 'Workflow$run()' / '$resume()', 'WorkflowAgent$invoke()' and 'LeadAgent$invoke()' / '$resume()' accept `stream = TRUE` and an optional `on_event` callback. While streaming, every unit of progress is delivered to the callback as a 'mini007_event' - a list with a 'type' discriminator - as soon as it happens, instead of only returning the final result at the end. The return value of the call is unchanged: the same final response you get without streaming.

Events from nested components are forwarded to the outermost callback, so a single 'on_event' passed to a 'LeadAgent' also receives the token deltas of every 'Agent' it delegates to, including Agents running inside a 'Workflow' wrapped with `$as_agent()`.

Event fields

‘type‘ ‘[character(1)]‘ The event type (see below).

‘source‘ ‘[character(1)]‘ Name of the Agent, Workflow or LeadAgent that emitted the event.

‘text‘ ‘[character(1) | NULL]‘ Text payload: a text delta for ‘"text_delta"‘ (see below), the full response or output for the ‘*_completed‘ and ‘*_output‘ events.

‘data‘ ‘[list]‘ Type-specific details (e.g. ‘tool_name‘, ‘arguments‘, ‘from_cache‘, ‘pending‘). Events are tagged with where they happened: ‘station‘ and ‘step‘ inside a Workflow, ‘task‘ (the plan index) and ‘agent‘ inside a LeadAgent. Tags from an inner component are kept, so a delta from an Agent in a Workflow wrapped as a LeadAgent sub-agent carries all four.

‘timestamp‘ ‘[POSIXct]‘ When the event was emitted.

Text deltas

An LLM produces its answer a few characters at a time. While streaming, each new fragment is delivered as soon as it is generated, in a ‘"text_delta"‘ event: a delta is the text that is new since the previous event, never the whole answer. Pasting the ‘text‘ of all deltas together, in order, gives the full response. ‘"thinking_delta"‘ events work the same way for the reasoning of reasoning models.

Event types

Agent ‘"agent_started"‘, ‘"text_delta"‘, ‘"thinking_delta"‘, ‘"tool_call"‘, ‘"tool_result"‘, ‘"agent_completed"‘.

Workflow ‘"workflow_started"‘, ‘"station_started"‘, ‘"station_retry"‘, ‘"station_fallback"‘, ‘"station_completed"‘, ‘"hitl_request"‘, ‘"workflow_output"‘, ‘"workflow_failed"‘.

LeadAgent ‘"plan_created"‘, ‘"task_started"‘, ‘"task_completed"‘, ‘"hitl_request"‘, ‘"lead_output"‘.

Cancellation

Raising an error from inside ‘on_event‘ (e.g. `stop("cancelled")`) stops the run immediately. It is re-raised as a condition of class ‘mini007_stream_cancelled‘, which Workflow retries and fallbacks never swallow, and the Agent’s message history is kept in sync with whatever the LLM produced before the cancellation.

Parallel execution

Stations in a parallel group run in separate ‘mirai‘ daemon processes, so their token deltas cannot be streamed back. They still emit ‘"station_started"‘ and ‘"station_completed"‘ events from the main process.

Examples

```
## Not run:
agent <- Agent$new(
  name = "poet",
  instruction = "You write short poems.",
  llm_object = ellmer::chat("openai/gpt-4.1-mini", echo = "none")
)

# Print tokens to the console as they arrive
```

```
agent$invoke("A haiku about R", stream = TRUE)

# Handle events yourself
agent$invoke("A haiku about R", on_event = function(event) {
  if (event$type == "text_delta") cat(event$text)
})

## End(Not run)
```

`mini007_pending` *Pending HITL Request*

Description

When a 'Workflow' or 'LeadAgent' is run with `hitl_mode = "pause"`, reaching a Human-In-The-Loop step does not block on `readline()`. Instead, `$run()` / `$invoke()` return immediately with a 'mini007_pending' object describing the paused step. Inspect it, then call `$resume()` on the same 'Workflow'/'LeadAgent' with the chosen `action` to continue execution.

This makes HITL usable outside an interactive console - e.g. from a Shiny app, a Plumber endpoint, or a batch job that persists the pending request and resumes it later.

`Workflow` *Workflow*

Description

An R6 class for building sequential multi-agent pipelines. A 'Workflow' is composed of **Stations** (processing units - an 'Agent', a 'WorkflowAgent', or any plain R function) connected by **Routes** (directed links, optionally gated by a condition function). Execution is always sequential: the output of one Station becomes the input of the next.

Stations whose results have already been computed can be retrieved from an internal cache instead of being re-executed, which is useful when iterating on later parts of the pipeline without paying the cost (or latency) of earlier LLM calls.

A finished 'Workflow' can be wrapped as a 'WorkflowAgent' via `$as_agent()`, making it composable with 'LeadAgent' or embeddable as a Station inside another 'Workflow'.

Public fields

`name` Workflow identifier.

`description` Optional human-readable description.

`use_cache` Whether to cache and reuse Station results.

`cache` Environment used as a hash-map for cached Station outputs.

`run_history` List of records from every `$run()` call.

`hitl_steps` Integer vector of step numbers at which execution pauses for human review. Set via `$set_hitl()`.

`hitl_mode` Either '"console"' (blocking `readline()` prompt, the default) or '"pause"' (non-blocking: `$run()` returns a 'mini007_pending' object instead of blocking). Set via `$set_hitl()`.

`n_daemons` Number of parallel daemons. Set via `$set_daemons()`.

Methods

Public methods:

- `Workflow$new()`
- `Workflow$add_station()`
- `Workflow$add_route()`
- `Workflow$set_entry()`
- `Workflow$run()`
- `Workflow$resume()`
- `Workflow$set_daemons()`
- `Workflow$add_parallel_group()`
- `Workflow$set_hitl()`
- `Workflow$clear_cache()`
- `Workflow$as_agent()`
- `Workflow$visualize()`

Method `new()`**:** Create a new 'Workflow'.

Usage:

```
Workflow$new(name, description = NULL, use_cache = TRUE)
```

Arguments:

`name` '[character(1)]' Workflow name.

`description` '[character(1)]' Optional description.

`use_cache` '[logical(1)]' Enable result caching (default 'TRUE').

Method `add_station()`**:** Add a Station to the workflow.

A Station is a named processing unit. Its 'handler' can be:

- An 'Agent' or 'WorkflowAgent' - the Station calls `handler$invoke(input)`.
- A plain R `function(input)` - called directly; return value is coerced to 'character'.

Usage:

```
Workflow$add_station(
  name,
  handler,
  description = NULL,
  max_retries = 0,
  retry_delay = 1,
  fallback = NULL
)
```

Arguments:

`name` '[character(1)]' Unique Station name within this workflow.

`handler` An 'Agent', 'WorkflowAgent', or 'function'.

`description` '[character(1)]' Optional human-readable description (shown in `$visualize()`).

`max_retries` '[integerish(1)]' Number of additional attempts after the first failure (default '0', i.e. no retries).

`retry_delay` '[numeric(1)]' Seconds to wait between retry attempts (default '1').

`fallback` '[function | NULL]' A `function(input, error)` invoked when all retry attempts are exhausted. 'NULL' re-raises the last error.

Returns: Invisibly returns 'self' for method chaining.

Method `add_route()`**:** Add a Route between two Stations.

Routes define the execution order. An optional 'condition' function receives the output of the 'from' Station and must return 'TRUE' or 'FALSE'. Conditional routes are evaluated first (in insertion order); the first matching one is followed. If none match, the first unconditional route from that Station is used as a default.

Usage:

```
Workflow$add_route(from, to, condition = NULL)
```

Arguments:

`from` '[character(1)]' Name of the source Station.

`to` '[character(1)]' Name of the destination Station.

`condition` '[function | NULL]' A function `function(output)` returning a single logical value. 'NULL' means "always follow this route" (i.e., unconditional).

Returns: Invisibly returns 'self' for method chaining.

Method `set_entry()`**:** Set the entry Station where execution begins.

If not called, `$run()` defaults to the first Station added.

Usage:

```
Workflow$set_entry(station_name)
```

Arguments:

`station_name` '[character(1)]' Name of the entry Station.

Returns: Invisibly returns 'self' for method chaining.

Method `run()`**:** Execute the workflow sequentially.

The 'input' string is passed to the entry Station. Each Station's output becomes the next Station's input. Execution stops when a Station has no outgoing Route (or no Route whose condition evaluates to 'TRUE').

When `use_cache = TRUE`, a Station whose (name, input) pair has been seen before returns the cached output without re-invoking the handler. Use `$clear_cache()` to force re-execution.

With `stream = TRUE` (or an 'on_event' callback), progress is delivered as 'mini007_event' objects while the workflow executes: station start/completion, retries, fallbacks, HITL requests, the final output, and the token deltas of every Agent Station (see `mini007_event`).

Usage:

```
Workflow$run(input, stream = FALSE, on_event = NULL)
```

Arguments:

`input` ‘[character(1)]‘ The initial prompt / payload.

`stream` ‘[logical(1)]‘ Whether to stream progress. Without ‘on_event‘, Agent token deltas are printed to the console.

`on_event` ‘[function | NULL]‘ Callback receiving each ‘mini007_event‘. Supplying it implies `stream = TRUE`. Raising an error inside it cancels the run.

Returns: ‘[character(1)]‘ The output of the last Station executed, or - if execution reaches an HITL step while `hitl_mode = "pause"` - a ‘mini007_pending‘ object. Use `is_pending()` to check.

Method `resume()`**:** Resume a Workflow paused at an HITL step.

Only relevant when `hitl_mode = "pause"` (see `$set_hitl()`). `$run()` returns a ‘mini007_pending‘ object when it reaches a paused HITL step; call `$resume()` with that object’s ‘request_id‘ and a chosen ‘action‘ to continue execution from that point.

Usage:

```
Workflow$resume(
  request_id,
  action = c("continue", "edit", "abort"),
  value = NULL,
  stream = FALSE,
  on_event = NULL
)
```

Arguments:

`request_id` ‘[character(1)]‘ The ‘request_id‘ from the ‘mini007_pending‘ object returned by `$run()`.

`action` ‘[character(1)]‘ One of ‘"continue"‘ (use the proposed output as-is), ‘"edit"‘ (replace it with ‘value‘), or ‘"abort"‘ (raise an error, stopping the workflow).

`value` ‘[character(1) | NULL]‘ Required when `action = "edit"`: the replacement output.

`stream` ‘[logical(1)]‘ Whether to stream the remaining execution (see `$run()`).

`on_event` ‘[function | NULL]‘ Callback receiving each ‘mini007_event‘. Supplying it implies `stream = TRUE`.

Returns: ‘[character(1)]‘ The final output, or another ‘mini007_pending‘ object if a further HITL step is reached.

Method `set_daemons()`**:** Configure parallel execution daemons using mirai.

Sets up persistent daemon processes for parallel station execution. When ‘n > 0‘, daemons are created via `mirai::daemons()`. Call with ‘n = 0‘ to shut down all daemons.

Usage:

```
Workflow$set_daemons(n = 0L)
```

Arguments:

`n` ‘[integerish(1)]‘ Number of daemon processes (default ‘0‘). Use number of cores for optimal CPU parallelism.

Returns: Invisibly returns ‘self‘ for method chaining.

Method `add_parallel_group()`**:** Define a group of Stations to execute in parallel.

All specified Stations receive the same input and execute concurrently via mirai daemons. Their outputs are collected and merged by the 'merge_fn' before passing to the next Station(s).

Requires '$set_daemons(n > 0)' to be called first.

Usage:

```
Workflow$add_parallel_group(from, stations, to = NULL, merge_fn = NULL)
```

Arguments:

`from` '[character(1)]' Name of the predecessor Station whose output feeds into the parallel group.

`stations` '[character]' Names of Stations to run in parallel. All must exist and not be the 'from' Station.

`to` '[character(1)]' Name of the successor Station that receives the merged result (optional).

`merge_fn` '[function]' A function `function(results)` that takes a named list of results (names are station names) and returns a single character string. Default concatenates results with newlines.

Returns: Invisibly returns 'self' for method chaining.

Method `set_hitl()`**:** Set Human-In-The-Loop (HITL) pause points.

When execution reaches a step whose number is listed in 'steps', it pauses and presents the human with three choices:

1. Continue with the Station's original output.
2. Edit the output manually before the next Station receives it.
3. Stop the workflow immediately (raises an error).

HITL only fires on fresh Station executions - cache hits are skipped. Steps are numbered from 1 in execution order, matching the step counter shown in `$run()` output. You can set multiple steps at once: `wf$set_hitl(c(1, 3))`.

With `mode = "console"` (default), a paused step blocks on `readline()` right there in `$run()` - unchanged from previous behavior. With `mode = "pause"`, `$run()` instead returns immediately with a 'mini007_pending' object describing the paused step; call `$resume()` with the chosen action to continue. This allows HITL to work outside an interactive console (e.g. Shiny, Plumber, or a batch job resuming later) and lets workflow state be inspected/persisted between steps.

Usage:

```
Workflow$set_hitl(steps, mode = c("console", "pause"))
```

Arguments:

`steps` '[integerish]' One or more step numbers (>= 1).

`mode` '[character(1)]' '"console"' (default) or '"pause"'.

Returns: Invisibly returns 'self' for method chaining.

Method `clear_cache()`**:** Remove all cached Station results.

Usage:

```
Workflow$clear_cache()
```

Returns: Invisibly returns 'self'.

Method `as_agent()`**:** Wrap this Workflow as a 'WorkflowAgent'.

The returned 'WorkflowAgent' exposes `$invoke(prompt)` and holds an 'agent_id', making it compatible with 'LeadAgent$register_agents()' and usable as a Station handler inside another 'Workflow'.

Usage:

```
Workflow$as_agent(name = NULL, instruction = NULL)
```

Arguments:

`name` '[character(1)]' Name for the 'WorkflowAgent'. Defaults to "`\{workflow name\} (agent)`".

`instruction` '[character(1)]' Instruction string describing the agent's role. Used by 'LeadAgent' for task-agent matching.

Returns: A 'WorkflowAgent' object.

Method `visualize()`**:** Render the workflow as a directed graph via DiagrammeR.

Stations are shown as rounded boxes. Conditional Routes are shown as dashed arrows labelled "cond". The entry Station is marked with a filled circle labelled "START".

Usage:

```
Workflow$visualize()
```

Returns: A 'DiagrammeR' / 'htmlwidget' object.

`WorkflowAgent` *WorkflowAgent*

Description

A lightweight, Agent-compatible wrapper around a 'Workflow'. Created via `Workflow$as_agent()`.

'WorkflowAgent' exposes the same interface that 'LeadAgent' expects from any agent (`name`, `instruction`, `agent_id`, `$invoke()`), so it can be:

- Registered with `LeadAgent$register_agents()`.
- Used as a Station handler inside another 'Workflow'.

Do not instantiate 'WorkflowAgent' directly - use `Workflow$as_agent()`.

Public fields

`name` Agent name (visible to 'LeadAgent' for task matching).

`instruction` System instruction describing the agent's role.

`agent_id` Unique identifier (UUID).

`workflow` The underlying 'Workflow' object.

`messages` Conversation history as a list of `list(role, content)` entries.

Methods

Public methods:

- `WorkflowAgent$new()`
- `WorkflowAgent$invoke()`
- `WorkflowAgent$reset_conversation_history()`

Method `new()`**:** Create a new 'WorkflowAgent'. Prefer `Workflow$as_agent()`.

Usage:

```
WorkflowAgent$new(name, instruction, workflow)
```

Arguments:

`name` '[character(1)]' Agent name.

`instruction` '[character(1)]' Instruction / system prompt.

`workflow` A 'Workflow' object.

Method `invoke()`**:** Run the underlying workflow with 'prompt' as input.

Not compatible with a wrapped 'Workflow' whose `hitl_mode` is '"pause"': a 'WorkflowAgent' is invoked synchronously from a 'Workflow' Station or a 'LeadAgent' step, both of which expect a single character result back, not a 'mini007_pending' object to resume later. Use `hitl_mode = "console"` (the default), or drive the wrapped workflow directly with `$workflow$run()` / `$resume()` if you need pause/resume HITL.

Usage:

```
WorkflowAgent$invoke(prompt, stream = FALSE, on_event = NULL)
```

Arguments:

`prompt` '[character(1)]' The user prompt / input payload.

`stream` '[logical(1)]' Whether to stream the wrapped workflow's progress (see `Workflow$run()`).

`on_event` '[function | NULL]' Callback receiving each 'mini007_event' emitted by the wrapped workflow and its Agents. Supplying it implies `stream = TRUE`.

Returns: '[character(1)]' The final output of the workflow.

Method `reset_conversation_history()`**:** Clear the conversation history stored in `$messages`.

Usage:

```
WorkflowAgent$reset_conversation_history()
```

Returns: Invisibly returns 'self'.

Index

Agent, 2

is_pending, 22, *50*

LeadAgent, 23

mini007::Agent, *23*
mini007_event, *4*, *28*, 45, *49*
mini007_pending, 47

Workflow, 47
WorkflowAgent, 52