Package 'FocusGroup'

September 12, 2026

Type Package

Title Focus Group Simulation Using LLM Agents

Version 0.5.2

Description Moderated focus group discussions simulated with large language models. Participants carry personas built from respondent records or labeled survey files. A session runs under a specified turn-taking rule, and one turn can be experimentally manipulated to measure its effect on the next.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

Depends R (>= 4.1.0)

Imports R6, dplyr, tidyr, rlang, haven, withr, LLMR (>= 0.8.8)

Suggests testthat (>= 3.0.0), ggplot2, quanteda, quanteda.textstats, topicmodels, tidytext, knitr, rmarkdown, pkgdown, shiny, bslib, DT, LLMR.shiny (>= 0.1.2)

VignetteBuilder knitr

URL https://github.com/asanaei/FocusGroup, https://asanaei.github.io/FocusGroup/

BugReports https://github.com/asanaei/FocusGroup/issues

NeedsCompilation no

Author Ali Sanaei [aut, cre]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2026-09-12 13:20:14 UTC

Contents

`analyze_focus_group` *Analyze Focus Group Results*

Description

Runs descriptive analyses offline. Thematic analysis and a model-generated summary are opt-in and run only when ‘config‘ is supplied.

Usage

```
analyze_focus_group(
  focus_group_result,
  num_topics = 5,
  include_plots = TRUE,
  speakers = c("participants", "all"),
  language = "en",
  config = NULL,
  .runner = NULL
)
```

Arguments

`focus_group_result`	A ‘FocusGroup‘ object or ‘focus_group_result‘.
`num_topics`	Integer number of topics for topic modeling.
`include_plots`	Logical. Attempt the four descriptive plots.
`speakers`	Whose turns the text statistics describe: ‘"participants"‘ (default) excludes the moderator’s scripted turns; ‘"all"‘ keeps them. The transcript itself always retains every turn.

`language`	Language of the transcript (default ‘"en"‘). The TF-IDF stop-word list and the readability formula are English-specific, so those components are skipped, with a note in ‘issues‘, for any other value.
`config`	Optional explicit ‘LLMR::llm_config‘. When supplied, thematic analysis and a model summary are generated.
`.runner`	Optional function used instead of live model calls for the two model analyses.

Value

A ‘focus_group_analysis‘ containing returned analysis components and an ‘issues‘ table.

Examples

```
transcript <- data.frame(
  speaker = c("Moderator", "P1", "P2"),
  text = c("What matters most?", "Cost matters.", "Access matters.")
)
fg <- focus_group_from_transcript(transcript)
analysis <- analyze_focus_group(fg, include_plots = FALSE)
analysis$basic_stats$speaker_stats
```

`ConversationFlow` *ConversationFlow Base Class*

Description

‘ConversationFlow‘ is an R6 base class defining the interface for turn-taking mechanisms in a focus group simulation. Subclasses implement specific strategies for selecting the next speaker.

Value

An R6 generator meant for subclassing; instances are built by [create_conversation_flow()], which returns a ‘ConversationFlow‘ subclass object ready for [FocusGroup].

Methods for Subclassing

‘initialize(agents, moderator_id)‘ Sets up the flow. Call ‘super$initialize(agents, moderator_id)‘.

‘select_next_speaker(focus_group)‘ *Required*. Logic to choose the next ‘FGAgent‘ to speak. Should return the agent object or ‘NULL‘.

‘update_state_post_selection(speaker_id, focus_group)‘ Optional. Updates internal state after a speaker has spoken. Base implementation updates ‘self$last_speaker_id‘.

Public fields

`agents` A named list of 'FGAgent' objects participating in the conversation.

`participant_ids` A character vector of agent identifiers, excluding the moderator.

`moderator_id` Character. The ID of the moderator agent.

`last_speaker_id` The ID of the agent who last spoke. Can be 'NULL'.

`selection_metadata` Details recorded for the latest selection. Desire-scoring failures record the condition and neutral fallback here.

Methods

Public methods:

- `ConversationFlow$new()`
- `ConversationFlow$select_next_speaker()`
- `ConversationFlow$update_state_post_selection()`
- `ConversationFlow$clone()`

Method `new()`**:** Initializes the ConversationFlow object.

Usage:

```
ConversationFlow$new(agents, moderator_id)
```

Arguments:

`agents` A named list of 'FGAgent' objects.

`moderator_id` Character. The ID of the moderator agent.

Method `select_next_speaker()`**:** Selects the next speaker. Must be implemented by subclasses.

Usage:

```
ConversationFlow$select_next_speaker(focus_group)
```

Arguments:

`focus_group` The 'FocusGroup' object managing the simulation, providing context.

Returns: The 'FGAgent' object of the selected speaker, or 'NULL'.

Method `update_state_post_selection()`**:** Updates internal state after a speaker is selected.

Usage:

```
ConversationFlow$update_state_post_selection(speaker_id, focus_group)
```

Arguments:

`speaker_id` The ID of the agent who was selected and just spoke.

`focus_group` The 'FocusGroup' object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
RandomFlow <- R6::R6Class("RandomFlow", inherit = ConversationFlow,
  public = list(
    select_next_speaker = function(focus_group) {
      ids <- self$participant_ids
      self$agents[[sample(ids, 1)]]
    }))
```

`create_agents` *Create Focus Group Agents*

Description

Creates participant agents from supplied records, direct personas, or the bundled ANES 2024 records, and adds a moderator.

Usage

```
create_agents(
  n_participants,
  config,
  demographics = NULL,
  survey_responses = NULL,
  direct_persona_descriptions = NULL,
  .runner = NULL
)
```

Arguments

`n_participants`	Integer number of participants to create
`config`	An explicit 'LLMR::llm_config' for all agents.
`demographics`	Optional data frame, one row per participant. When it, 'survey_responses', and 'direct_persona_descriptions' are all NULL, participants are drawn from the bundled ANES 2024 records.
`survey_responses`	Optional data frame of survey answers, one row per participant.
`direct_persona_descriptions`	Optional character vector of pre-rendered participant personas. When supplied, these descriptions are used directly and are recycled in order if necessary. Demographics and survey responses remain attached as raw reporting fields.
`.runner`	Optional function used instead of live model calls and stored on every agent.

Value

A named list of FGAgent objects (participants + 1 moderator), keyed by agent ID.

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
agents <- create_agents(6, config = cfg)
names(agents)
agents$P1$persona_description

demo_data <- data.frame(
  age = c(22, 35, 28, 41, 19, 33),
  gender = c("Female", "Male", "Male", "Female", "Male", "Female"),
  education = c("Bachelor's", "Master's", "High School", "PhD",
                "Some College", "Bachelor's")
)
agents <- create_agents(6, config = cfg, demographics = demo_data)
agents$P2$persona_description
```

`create_agents_from_data` *Create agents from an in-memory data frame of respondents*

Description

Like [create_agents_from_survey()] but starting from a data frame already in memory (for example ‘LLMR::anes_2024_personas‘). Demographic columns are rendered as background; the remaining columns are rendered as survey responses, keyed by their question wording when the frame carries a ‘dictionary‘ attribute (see [LLMR::llm_persona_split()]), else by their column names. Values are taken as-is (decode and clean them first if they are still coded).

Usage

```
create_agents_from_data(
  data,
  n_participants,
  config,
  demographic_cols = NULL,
  rows = NULL,
  weights = NULL,
  .runner = NULL
)
```

Arguments

`data`	A data frame, one respondent per row.
`n_participants`	Integer number of participants (excludes the moderator).
`config`	An explicit ‘LLMR::llm_config‘ for all agents.

demographic_cols
: Character vector of columns to render as demographics. Defaults to the ‘data‘’s ‘"demographic_fields"‘ attribute when present, else a small set of common demographic column names found in ‘data‘.

rows, weights
: See [create_agents_from_survey()].

.runner
: Optional function used instead of live model calls and stored on every agent.

Details

A frame of class ‘silicon_panel‘, or a frame with both ‘persona‘ and ‘persona_id‘ columns, is treated as a pre-rendered persona panel. Its ‘persona‘ text becomes each participant’s direct persona description; ‘persona‘ and ‘persona_id‘ are not rendered as survey answers. This bridge uses the frame’s structure and does not require LLMRpanel.

Value

A named list of ‘FGAgent‘ objects (participants + moderator), keyed by agent ID.

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
agents <- create_agents_from_data(
  LLMR::anes_2024_personas, n_participants = 6, config = cfg
)
agents$P1$persona_description
```

---

create_agents_from_survey

Create agents from a labeled survey file

Description

Reads a labeled survey file (Stata ‘.dta‘, SPSS ‘.sav‘, or SAS ‘.sas7bdat‘), decodes the chosen variables from their value labels, and turns each selected respondent into an ‘FGAgent‘ whose persona states that respondent’s demographics and survey answers. Numeric codes are decoded from the file’s own value labels, so the same call works on ANES, GSS, WVS, or any other labeled file; nothing about a particular dataset is hard-coded.

Usage

```
create_agents_from_survey(
  n_participants,
  survey_path,
  config,
  demographic_vars = NULL,
  survey_vars = NULL,
  rows = NULL,
```

```
  weights = NULL,
  na_strings = .fg_default_na_strings,
  .runner = NULL
)
```

Arguments

n_participants	Integer number of participants (excludes the moderator).
survey_path	Path to the survey file ('.dta', '.sav', or '.sas7bdat').
config	An explicit 'LLMR::llm_config' for all agents.
demographic_vars	Variable names (codes) to render as demographics. May be a named vector, in which case the names are shown as the field labels. If 'NULL', common demographic variables are auto-detected by their labels.
survey_vars	Variable names (codes) to render as survey responses. May be named (names become the question wording shown). If 'NULL', labeled variables that are not demographics are used.
rows	Optional row selector restricting the eligible respondents before sampling: an integer or logical vector, or a predicate 'function(df) -> logical' over the decoded demographics frame.
weights	Optional sampling weights: a column name in the file, or a numeric vector aligned to the file's rows. Used only to weight which respondents are drawn.
na_strings	Character vector of value-label substrings treated as missing (case-insensitive). Defaults to a small common set; pass your own to match another file's missing-data vocabulary.
.runner	Optional function used instead of live model calls and stored on every agent.

Details

The survey answers are keyed by each variable's question wording (its label in the file) so the model sees the item, not a code name. The persona draws no inferences from the answers; it states them and lets the model interpret.

Value

A named list of 'FGAgent' objects (participants + moderator), keyed by agent ID.

See Also

[create_agents_from_data()] for an in-memory data frame, and 'LLMR::anes_2024_personas' for a ready-made example.

Examples

```
# A small labeled file stands in for a survey release; the call is the same
# against a full ANES, GSS, or WVS file.
toy <- data.frame(
  age = haven::labelled(c(1L, 2L, 3L, 2L),
```

```
                          c("18-24" = 1L, "25-34" = 2L, "35-44" = 3L)),
  pid = haven::labelled(c(1L, 2L, 2L, 1L),
                          c("Democrat" = 1L, "Republican" = 2L))
)
path <- file.path(tempdir(), "toy_survey.dta")
haven::write_dta(toy, path)

agents <- create_agents_from_survey(
  n_participants = 2,
  survey_path = path,
  config = LLMR::llm_config("groq", "openai/gpt-oss-20b",
                            api_key = LLMR::llm_api_key_env("GROQ_API_KEY")),
  demographic_vars = c(age = "age"),
  survey_vars = c("Party identification" = "pid"),
  rows = function(df) df$age != "18-24"
)
agents$P1$persona_description
```

`create_conversation_flow` *Create a Conversation Flow Object*

Description

Factory function to create an instance of a ‘ConversationFlow‘ subclass.

Usage

```
create_conversation_flow(mode, agents, moderator_id, flow_params = list())
```

Arguments

`mode`	Character. The type of turn-taking flow to create. Supported: "round_robin", "probabilistic", "desire_based".
`agents`	A named list of ‘FGAgent‘ objects.
`moderator_id`	Character. The ID of the moderator agent.
`flow_params`	List. Additional parameters specific to the chosen flow type. For "probabilistic": ‘initial_propensities‘ (named numeric vector), ‘recovery_increment‘ (numeric). For "desire_based": ‘min_desire_threshold‘ (numeric).

Value

An initialized ‘ConversationFlow‘ object (e.g., ‘RoundRobinFlow‘, ‘ProbabilisticFlow‘, ‘Desire-BasedFlow‘).

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
agents <- create_agents(
  n_participants = 2, config = cfg,
  direct_persona_descriptions = c("A retired teacher.", "A night-shift nurse."))
flow <- create_conversation_flow("desire_based", agents, "MOD",
                                 flow_params = list(min_desire_threshold = 4))
class(flow)
flow$participant_ids
```

FGAgent *FGAgent Class*

Description

FGAgent Class

FGAgent Class

Details

Represents an agent (participant or moderator) in a focus group simulation. Each agent has a unique ID, a persona, an LLM configuration, and methods to generate utterances and express a desire to speak.

Value

An R6 generator. ‘FGAgent$new(id, agent_details, config, is_moderator = FALSE, .runner = NULL)‘ returns an ‘FGAgent‘; [create_agents()] and its relatives construct whole rosters.

Customizing Agents

‘FGAgent‘ is designed to be flexible:

- **Per-Agent LLM Configuration**: Each agent is initialized with its own ‘llm_config‘ (stored in the ‘config‘ field). This allows different agents to use different LLMs, temperatures, or even providers.
- **Persona Definition**: The ‘agent_details‘ (containing ‘demographics‘, ‘survey_responses‘, ‘direct_persona_description‘, and/or ‘communication_style‘) are used to construct the ‘persona_description‘ and ‘communication_style_instruction‘.
- **Subclassing**: For advanced customization, inherit from ‘FGAgent‘ to override methods or add new fields.

Public fields

`id` Character. Unique identifier for the agent.

`persona_description` Character. Textual description of the agent's persona, derived from demographics, survey responses, or direct input.

`communication_style_instruction` Character. A specific instruction about the agent's communication style, to be included in prompts.

`config` An 'llm_config' object (from the 'LLMR' package) specifying the LLM provider, model, API key, and other parameters for this agent.

`.runner` 'NULL' or a function used instead of live model calls. It receives a data frame with 'config' and 'messages' list-columns and returns those rows with at least 'response_text'.

`is_moderator` Logical. 'TRUE' if the agent is the moderator, 'FALSE' otherwise.

`history` List. A log of utterances made by this agent during the simulation.

`tokens_sent_agent` Numeric. Total tokens sent by this agent.

`tokens_received_agent` Numeric. Total tokens received by this agent.

`role` Character. "moderator" or "participant" for convenience in reports.

`demographics` Named list. Raw demographics used to build persona.

`survey_responses` Named list. Raw survey responses used to build persona.

Methods

Public methods:

- `FGAgent$new()`
- `FGAgent$generate_utterance()`
- `FGAgent$get_need_to_talk()`
- `FGAgent$clone()`

Method `new()`**:** Initialize a new FGAgent.

Usage:

```
FGAgent$new(id, agent_details, config, is_moderator = FALSE, .runner = NULL)
```

Arguments:

`id` Character. A unique identifier for the agent.

`agent_details` List. Contains information to build the agent's persona. Can include:

- 'demographics': A named list of demographic attributes (e.g., 'list(age = 30, occupation = "teacher")').
- 'survey_responses': A named list of survey questions and answers.
- 'direct_persona_description': A character string to be used directly as the persona. Overrides demographics/survey if provided.
- 'communication_style': A character string describing the agent's communication style (e.g., "analytical and direct", "empathetic and story-driven").

If 'is_moderator' is 'TRUE' and no specific details are provided, a default moderator persona is used.

`config` An 'llm_config' object from 'LLMR::llm_config()'. It may be 'NULL' only for an analysis-only agent that will not generate output.

`is_moderator` Logical. 'TRUE' if this agent is the moderator, 'FALSE' otherwise.

`.runner` 'NULL' uses live LLMR calls; a function is used instead of live model calls.

Method `generate_utterance()`**:** Generate an utterance for the agent.

Usage:

```
FGAgent$generate_utterance(
  topic,
  conversation_history_string,
  utterance_prompt_template,
  max_tokens_utterance = 150,
  current_moderator_question = "N/A",
  conversation_summary_so_far = "N/A",
  current_phase = "discussion",
  conversation_log = NULL,
  standing_rules = NULL,
  self_state = TRUE
)
```

Arguments:

`topic` Character. The current discussion topic.

`conversation_history_string` Character. Formatted string of recent conversation history.

`utterance_prompt_template` Character. The prompt template to use.

`max_tokens_utterance` Integer. Maximum tokens for the generated utterance.

`current_moderator_question` Character. The current question posed by the moderator.

`conversation_summary_so_far` Character. A summary of earlier parts of the conversation.

`current_phase` Character. The current phase of the focus group (e.g., "icebreaker", "exploration").

`conversation_log` List or NULL. The structured conversation log. When supplied (and the template is not a legacy flat template), the message is built role-flipped: this agent's own prior turns become 'assistant' messages and others' become labeled 'user' messages. 'NULL' keeps the legacy flat single-user-message construction.

`standing_rules` Character or NULL. The system-message standing rules (persona-anchoring, safety, etc.) for the role-flipped path; 'NULL' uses a built-in default appropriate to the agent's role.

`self_state` Logical. If 'TRUE' (default), a compact "points you have already made" digest of the agent's own prior turns is added to the system message (participants only) to discourage self-repetition.

Returns: A list with 'text' (the generated utterance) and 'meta' (a list of call metadata: token counts, finish reason, provider, model, timing).

Method `get_need_to_talk()`**:** Get the agent's "desire to talk" score. This method queries the LLM to rate how strongly the agent feels the need to contribute to the discussion at the current moment.

Usage:

```
FGAgent$get_need_to_talk(
  topic,
  conversation_history_string,
  desire_prompt_template,
  max_tokens_desire = 20,
  current_moderator_question = "N/A",
  last_speaker_id = "N/A",
  last_utterance_text = "N/A",
  conversation_log = NULL
)
```

Arguments:

`topic` Character. The current discussion topic.

`conversation_history_string` Character. Formatted string of recent conversation history.

`desire_prompt_template` Character. The prompt template to use for this query.

`max_tokens_desire` Integer. Maximum tokens for the LLM's response to the desire query.

`current_moderator_question` Character. The current question posed by the moderator.

`last_speaker_id` Character. The ID of the agent who spoke last.

`last_utterance_text` Character. The text of the last utterance.

`conversation_log` List or NULL. The structured conversation log. When supplied (and the template is not a legacy flat template), desire scoring is role-flipped so the agent reads its own prior turns as its own voice; 'NULL' keeps the legacy flat construction.

Returns: Numeric. A score from 0 (no desire) to 10 (very strong desire).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
a <- FGAgent$new(
  id = "P1",
  agent_details = list(direct_persona_description = "A retired teacher."),
  config = cfg)
a$persona_description
```

`FocusGroup` *FocusGroup R6 Class*

Description

Main R6 class for managing and running a focus group simulation. It orchestrates the simulation through distinct phases (Opening, Icebreaker, Engagement, Exploration, Closing) guided by a question script, manages agent interactions via a turn-taking flow, and provides methods for analysis and visualization of the conversation.

Value

An R6 generator. 'FocusGroup$new()' returns a session object; most analyses run through [run_focus_group()], which builds, runs, and returns one. Construction alone makes no model call.

Public fields

`topic` Character. The main topic of the focus group discussion.

`purpose` Character. The primary purpose or research objective of conducting the focus group.

`agents` Named list. A list of 'FGAgent' objects participating in the simulation, indexed by their IDs.

`moderator_id` Character. The ID of the agent designated as the moderator.

`conversation_log` List. A chronological log of all messages. Each message is a list that includes at least 'message_id', 'round', 'speaker_id', 'is_moderator', 'text', 'timestamp', 'phase', and 'metadata', along with call metadata: 'response_id', 'finish_reason', 'sent_tokens', 'rec_tokens', 'total_tokens', 'duration_s', 'provider', and 'model'. 'message_id' is unique message order; moderator and participant messages from one moderator cycle share 'round'.

`turn_taking_flow` A 'ConversationFlow' object dictating participant turn-taking.

`prompt_templates` List. Holds prompt templates for agent/moderator actions.

`question_script` List. A structured list defining phases and specific questions/actions for the moderator. Each element is a list with 'phase' (e.g., "opening", "icebreaker_question") and optionally 'text' (for specific questions).

`current_question_text` Character. Text of the current question being discussed.

`current_conversation_summary` Character. An LLM-generated summary of earlier parts of the conversation, used for managing context length in prompts.

`final_summary` Character. Final LLM-generated summary from the most recent simulation run.

`message_mode` Character. The message construction used by the most recent 'run_simulation()' ("roleflip" or "flat"), recorded so a saved object can be replayed (e.g. in the GUI continuation experiment) with the same construction.

`admin_config` An 'llm_config' object for group-level model tasks.

`max_tokens_utterance` Integer. Default max tokens for participant utterances.

`max_tokens_moderator` Integer. Default max tokens for moderator utterances.

`max_tokens_desire` Integer. Default max tokens for desire-to-talk queries.

`max_participant_responses` Integer. Maximum number of participant exchanges per round before the moderator can intervene. Can also be set globally via 'options(focusgroup.max_participant_responses = N)'.

`total_tokens_sent` Numeric. Total tokens sent across all LLM calls in the group.

`total_tokens_received` Numeric. Total tokens received across all LLM calls.

Methods

Public methods:

- FocusGroup$new()
- FocusGroup$run_simulation()
- FocusGroup$summarize()
- FocusGroup$analyze()
- FocusGroup$analyze_topics()
- FocusGroup$analyze_tfidf()
- FocusGroup$analyze_readability()
- FocusGroup$analyze_themes()
- FocusGroup$analyze_statistics()
- FocusGroup$analyze_participation_balance()
- FocusGroup$analyze_response_patterns()
- FocusGroup$analyze_question_patterns()
- FocusGroup$analyze_key_phrases()
- FocusGroup$plot_participation_timeline()
- FocusGroup$plot_word_count_distribution()
- FocusGroup$plot_participation_by_agent()
- FocusGroup$plot_message_length_timeline()
- FocusGroup$clone()

Method `new()`**:** Initialize a new FocusGroup simulation.

Usage:

```
FocusGroup$new(
  topic,
  purpose,
  agents,
  moderator_id,
  turn_taking_flow,
  question_script = list(),
  prompt_templates = list(),
  admin_config = NULL,
  max_tokens_config = list(),
  max_participant_responses = NULL
)
```

Arguments:

`topic` Character. The main discussion topic.

`purpose` Character. The primary purpose of the focus group.

`agents` Named list of initialized 'FGAgent' objects.

`moderator_id` Character. The ID of the agent acting as moderator.

`turn_taking_flow` An initialized 'ConversationFlow' object.

`question_script` List. Moderator's script defining phases and questions. If empty, a minimal default script (opening, generic discussion, closing) is used.

`prompt_templates` List. Custom prompt templates. Defaults are used if not provided.

`admin_config` An 'llm_config' object for administrative model tasks. If 'NULL', the moderator's explicitly supplied 'config' is used.

`max_tokens_config` List. Optional. Named list with 'utterance', 'moderator', 'desire' to override default max token limits for these LLM call types.

`max_participant_responses` Integer. Optional. Maximum participant exchanges per round before moderator intervention. Defaults to 'getOption("focusgroup.max_participant_responses", 3)'.

Method `run_simulation()`**:** Run the full focus group simulation. Iterates through the 'question_script' phases or a specified number of rounds.

Usage:

```
FocusGroup$run_simulation(num_rounds = NULL, verbose = FALSE)
```

Arguments:

`num_rounds` Integer. Optional. Maximum number of moderator cycles to run. If 'NULL' (default), the simulation runs until the 'question_script' is exhausted or the moderator decides to end. If both 'num_rounds' and 'question_script' are provided, the simulation stops at whichever condition is met first.

`verbose` Logical. If 'TRUE', prints progress and utterances to the console.

Returns: Invisibly returns the 'conversation_log'.

Method `summarize()`**:** Generate a summary of the conversation using an LLM.

Usage:

```
FocusGroup$summarize(
  config,
  summary_level = 1,
  max_tokens = NULL,
  internal_call = FALSE,
  transcript_override = NULL,
  .runner = NULL
)
```

Arguments:

`config` An explicit 'llm_config' object for the summarization model.

`summary_level` Integer (1-3). 1: Prose overview, 2: Detailed bulleted, 3: Short bulleted takeaways.

`max_tokens` Integer. Optional. Max tokens for the summary.

`internal_call` Logical. If TRUE, this is an internal call (e.g. for context window management) and token counts are not added to the agent who owns 'admin_config'.

`transcript_override` Character. Optional. If provided, this transcript is summarized instead of 'self$conversation_log'.

`.runner` Optional function used instead of live model calls. It receives a data frame with 'config' and 'messages' list-columns and returns the rows with at least 'response_text'.

Returns: Character string containing the generated summary.

Method `analyze()`**:** Basic analysis of the conversation log.

Usage:

```
FocusGroup$analyze(message_ids = NULL, speaker_ids = NULL)
```

Arguments:

`message_ids` Integer vector. Optional ‘message_id‘ values to analyze. If ‘NULL‘, all messages are analyzed.

`speaker_ids` Character vector. Optional. Specific speakers to analyze. If ‘NULL‘, analyzes all speakers.

Returns: A list with ‘speaker_stats‘ (a tibble: speaker_id, utterance_count, total_words, avg_words_per_utterance) and ‘full_transcript‘ (character string).

Method `analyze_topics()`**:** Perform LDA topic modeling on the conversation.

Usage:

```
FocusGroup$analyze_topics(
  num_topics = 5,
  min_doc_length = 20,
  top_n_terms = 10,
  message_ids = NULL,
  speaker_ids = NULL,
  seed = 110,
  unit = c("message", "speaker"),
  ...
)
```

Arguments:

`num_topics` Integer. Number of topics to identify.

`min_doc_length` Integer. Minimum words for a text to count as a document.

`top_n_terms` Integer. Number of top terms per topic to return.

`message_ids` Integer vector. Optional ‘message_id‘ values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

`seed` Integer or NULL. Seed for the LDA control (default 110); ‘NULL‘ leaves it unset.

`unit` Character. What counts as a document: ‘"message"‘ (default) treats each utterance as a document; ‘"speaker"‘ pools every turn by the same speaker, which leaves as many documents as there are speakers and tends to recover who spoke rather than what was discussed.

`...` Additional arguments to ‘topicmodels::LDA()‘.

Returns: A list with LDA model, topic terms, and document-topic proportions. ‘NULL‘ on failure.

Method `analyze_tfidf()`**:** Calculate TF-IDF scores for terms per participant.

Usage:

```
FocusGroup$analyze_tfidf(
  top_n_terms = 10,
  message_ids = NULL,
  speaker_ids = NULL,
  ...
)
```

Arguments:

`top_n_terms` Integer. Number of top TF-IDF terms per participant.

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

`...` Additional arguments to 'tidytext::unnest_tokens'.

Returns: A tibble with TF-IDF scores.

Method `analyze_readability()`**:** Calculate readability scores for each participant's aggregated text.

Usage:

```
FocusGroup$analyze_readability(
  measures = "Flesch",
  message_ids = NULL,
  speaker_ids = NULL
)
```

Arguments:

`measures` Character vector. Readability measure(s) from 'quanteda.textstats::textstat_readability'.

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A tibble with readability scores.

Method `analyze_themes()`**:** Perform LLM-assisted thematic analysis on the transcript.

Usage:

```
FocusGroup$analyze_themes(
  config,
  message_ids = NULL,
  speaker_ids = NULL,
  .runner = NULL
)
```

Arguments:

`config` An explicit 'llm_config' object for thematic analysis.

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

`.runner` Optional function used instead of live model calls. It receives a data frame with 'config' and 'messages' list-columns and returns the rows with at least 'response_text'.

Returns: The thematic summary as a character string. An empty transcript returns 'character(0)'. Provider failures are propagated.

Method `analyze_statistics()`**:** Perform statistical analysis on conversation patterns.

Usage:

```
FocusGroup$analyze_statistics(message_ids = NULL, speaker_ids = NULL)
```

Arguments:

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A list with ANOVA results, phase participation stats, and correlations.

Method `analyze_participation_balance()`**:** Analyze participation balance and dominance patterns.

Usage:

```
FocusGroup$analyze_participation_balance(
  message_ids = NULL,
  speaker_ids = NULL
)
```

Arguments:

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A list with participation statistics and balance metrics.

Method `analyze_response_patterns()`**:** Analyze response patterns and interaction behaviors.

Usage:

```
FocusGroup$analyze_response_patterns(message_ids = NULL, speaker_ids = NULL)
```

Arguments:

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A list with response and interaction pattern metrics.

Method `analyze_question_patterns()`**:** Analyze question asking patterns during the conversation.

Usage:

```
FocusGroup$analyze_question_patterns(message_ids = NULL, speaker_ids = NULL)
```

Arguments:

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A list with question pattern analysis.

Method `analyze_key_phrases()`**:** Extract and analyze key phrases using n-grams.

Usage:

```
FocusGroup$analyze_key_phrases(
  min_freq = 2,
  message_ids = NULL,
  speaker_ids = NULL
)
```

Arguments:

`min_freq` Integer. Minimum frequency for phrases to be considered key.

`message_ids` Integer vector. Optional 'message_id' values to analyze.

`speaker_ids` Character vector. Optional. Specific speakers to analyze.

Returns: A list with bigram and trigram analysis.

Method `plot_participation_timeline()`**:** Create a participation timeline of cumulative messages by participant across phases.

Usage:

```
FocusGroup$plot_participation_timeline()
```

Returns: ggplot object

Method `plot_word_count_distribution()`**:** Create word count distribution plot showing message length patterns.

Usage:

```
FocusGroup$plot_word_count_distribution()
```

Returns: ggplot object

Method `plot_participation_by_agent()`**:** Create a participation by agent plot showing total messages per participant.

Usage:

```
FocusGroup$plot_participation_by_agent()
```

Returns: ggplot object

Method `plot_message_length_timeline()`**:** Create a timeline showing message length over message order.

Usage:

```
FocusGroup$plot_message_length_timeline()
```

Returns: ggplot object

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
agents <- create_agents(
  n_participants = 2, config = cfg,
  direct_persona_descriptions = c("A retired teacher.",
                                  "A night-shift nurse."))
flow <- create_conversation_flow("round_robin", agents, "MOD")
fg <- FocusGroup$new(topic = "Neighborhood safety",
                     purpose = "Pilot the moderator guide.",
                     agents = agents,
                     moderator_id = "MOD",
                     turn_taking_flow = flow)
fg$topic
```

`focus_group_from_transcript` *Build a FocusGroup object from an imported transcript*

Description

Turns a transcript that the package did not generate – a human focus group, a transcript exported from other software, or a saved CSV – into a [FocusGroup] object whose ‘conversation_log‘ holds the imported turns. The returned object supports the analysis methods (‘analyze()‘, ‘analyze_participation_balance()‘, ‘analyze_topics()‘, ‘analyze_tfidf()‘, ‘analyze_readability()‘, ‘analyze_themes()‘), the plotting methods, and [analyze_focus_group()]. No model call is made: importing and the descriptive analyses run offline, and only the model-dependent methods (‘analyze_themes()‘, ‘summarize()‘) contact a provider when you call them.

Usage

```
focus_group_from_transcript(
  data,
  speaker_col = "speaker",
  text_col = "text",
  topic = NULL,
  moderator_id = NULL
)
```

Arguments

`data`	A data frame with one row per turn, in order.
`speaker_col`	Character. Name of the column holding the speaker id. Default "speaker".
`text_col`	Character. Name of the column holding the utterance text. Default "text".
`topic`	Character or ‘NULL‘. The discussion topic, used by the model-dependent analyses and stored on the object. ‘NULL‘ falls back to "Imported transcript".
`moderator_id`	Character or ‘NULL‘. Exact speaker id to treat as the moderator. When ‘NULL‘, the documented "mod" substring fallback is used.

Details

One lightweight agent is created per speaker so that speaker-aware methods have a roster to report on; these agents carry a minimal placeholder persona and are not meant to continue the conversation. Set ‘moderator_id‘ when the transcript identifies the moderator. When it is ‘NULL‘, the moderator is the first speaker whose id contains "mod" (case-insensitive), e.g. "Moderator" or "MOD". When the fallback finds no match, a non-speaking "MOD" agent is added so the object is complete, and it appears in ‘analyze()‘ with zero utterances. Rows whose speaker is missing or blank are dropped with a warning; rows with speaker "System" are kept in the log but excluded from analyses, matching how the simulator treats its own roster message.

Value

A [FocusGroup] object with a populated ‘conversation_log‘.

See Also

[analyze_focus_group()] to run the descriptive analyses in one call; [run_focus_group()] to simulate a discussion instead.

Examples

```
transcript <- data.frame(
  speaker = c("Moderator", "Ana", "Ben", "Ana", "Moderator", "Ben"),
  text = c(
    "Welcome, everyone. What do you make of the new library hours?",
    "They help working parents like me; evenings matter most.",
    "I see it differently. Mornings are now far too crowded.",
    "Crowded mornings still beat being locked out after work.",
    "Ben, say more about what changed in the mornings.",
    "Staff are stretched thin before noon, so lines are longer."
  )
)
fg <- focus_group_from_transcript(transcript, topic = "library hours")
fg$analyze_participation_balance()$participation_stats
```

---

get_default_prompt_templates

Get Default Prompt Templates

Description

Retrieves a list containing the default prompt templates used by the ‘FocusGroup‘ package. These templates guide the behavior of LLM agents (participants and moderator) during different phases and actions within the simulation.

Usage

```
get_default_prompt_templates()
```

Details

Users can retrieve this list, modify specific templates, and then pass the modified list to ‘FocusGroup$new()‘ or higher-level wrapper functions to customize the simulation’s prompts.

Value

A named list where each element is a character string representing a prompt template. Placeholders like ‘{{topic}}‘, ‘{{persona_description}}‘, etc., are used within the templates and will be filled dynamically during the simulation.

Participant templates and message construction

Two participant templates ship here, and which one is used depends on the message construction (see the ‘message_mode‘ argument of [run_focus_group()], or ‘options(focusgroup.message_mode=)‘).

- ‘participant_turn_instruction‘ is the canonical template for the default role-flipped construction. It carries only the current question and the turn cue, because the persona, standing rules, and the transcript are supplied structurally (a system message plus role-separated turns).
- ‘participant_utterance_subtle_persona‘ is the flat template. It inlines the persona and the whole transcript into one user message. A custom template that contains ‘{{conversation_history}}‘ or ‘{{persona_description}}‘ is treated as a flat template and routed through the flat path.

Examples

```
default_prompts <- get_default_prompt_templates()
names(default_prompts)
default_prompts$participant_utterance_subtle_persona

my_prompts <- get_default_prompt_templates()
my_prompts$moderator_opening <- "Welcome all! Let us talk about {{topic}} today."

cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
agents <- create_agents(2, config = cfg,
                        direct_persona_descriptions = c("A retired teacher.",
                                                        "A night-shift nurse."))
fg <- FocusGroup$new(
  topic = "Library hours",
  purpose = "Pilot the revised opening.",
  agents = agents,
  moderator_id = "MOD",
  turn_taking_flow = create_conversation_flow("round_robin", agents, "MOD"),
  prompt_templates = my_prompts)
fg$prompt_templates$moderator_opening
```

---

run_focus_group *Run a Focus Group Simulation*

Description

Constructs the agents, moderator guide, and turn-taking flow, then runs one focus group session. ‘config‘ has no default; every agent and group-level call uses the one supplied. Before a live run, the function estimates the number of generated model outputs and applies the ‘max_calls‘ limit unless ‘confirm‘ is ‘TRUE‘.

Usage

```
run_focus_group(
  topic,
```

```
  config,
  n_participants = 6,
 guide = c(Opening = 1, Icebreaker = 1, Engagement = 2, Exploration = 3, Closing = 1),
  demographics = NULL,
  survey_responses = NULL,
  flow = "desire_based",
  seed = NULL,
  message_mode = c("roleflip", "flat"),
  verbose = TRUE,
  max_participant_responses = NULL,
  max_calls = 100L,
  confirm = FALSE,
  .runner = NULL
)
```

Arguments

topic	Character. Focus group topic.
config	An explicit ‘LLMR::llm_config‘ used by all agents and group-level model tasks.
n_participants	Integer. Number of participants, excluding the moderator.
guide	A named numeric vector or named list. Numeric values select that many moderator instructions from the phase banks. Character vectors supply the ordered instructions directly. The default is a compact session (one opening, one icebreaker, two engagement and three exploration questions, one closing); with the default flow, participants, and response limit it plans 93 model outputs, inside the default ‘max_calls‘ of 100. Larger guides multiply quickly: each question round costs one output per response plus, under ‘"desire_based"‘, one scoring output per participant.
demographics	Optional participant demographics, one row per participant. With this, ‘survey_responses‘, and personas all absent, the participants are drawn from the bundled ANES 2024 records.
survey_responses	Optional participant survey answers, one row per participant.
flow	Character. One of ‘"round_robin"‘, ‘"probabilistic"‘, or ‘"desire_based"‘.
seed	Optional integer governing in-package sampling.
message_mode	Character. ‘"roleflip"‘ or ‘"flat"‘.
verbose	Logical. Print session progress.
max_participant_responses	Optional integer maximum number of participant responses per moderator question (default 2).
max_calls	Integer. Maximum estimated live model outputs allowed without confirmation.
confirm	Logical. Permit a live run whose estimate exceeds ‘max_calls‘.
.runner	Optional function used instead of live model calls. It receives a data frame with ‘config‘ and ‘messages‘ list-columns and returns those rows with at least ‘response_text‘.

Value

A 'focus_group_result' with the group, transcript, summary, participant table, token usage, and sanitized metadata.

Examples

```
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                        api_key = LLMR::llm_api_key_env("GROQ_API_KEY"))
guide <- list(
  Opening = "Welcome the participants and state the ground rules.",
  Exploration = "Which funding priority deserves attention first?",
  Closing = "Thank the participants and close the session."
)

scripted <- function(experiments, ...) {
  experiments$response_text <- "Evening opening hours matter most to me."
  experiments
}
result <- run_focus_group(
  topic = "Library funding priorities",
  config = cfg,
  n_participants = 2,
  guide = guide,
  flow = "round_robin",
  verbose = FALSE,
  .runner = scripted
)
result$transcript[, c("speaker_id", "phase", "text")]


if (nzchar(Sys.getenv("GROQ_API_KEY"))) {
  live <- run_focus_group(
    topic = "Library funding priorities",
    config = cfg,
    n_participants = 4,
    guide = guide,
    flow = "round_robin"
  )
  live$summary
}
```

`run_focus_studio` *Launch the FocusGroup Shiny GUI*

Description

A point-and-click front end with three tabs. Run a focus group starts a fresh moderated session live from a topic and a handful of participants, shows the transcript, and offers a complete `.rds` session

and transcript CSV. Analyze accepts a finished run, saved session, CSV, or bundled example and runs the package's descriptive analyses offline; thematic analysis and a model summary are optional live actions. The continuation experiment takes a saved focus group up to a message and compares repeated next-message responses under one control history and several labeled treatment histories. Running and the continuation experiment generate text and need an API key; analysis works offline.

Usage

```
run_focus_studio(...)
```

Arguments

`...`	Passed to `shiny::runApp()`.

Details

The GUI is optional. It needs the suggested packages shiny, bslib, DT, and LLMR.shiny; install them first. Keys are read from environment variables only, never pasted into the app.

Value

Invisibly, the value of `shiny::runApp()`; called for the side effect of starting the app.

Examples

```
if (interactive() &&
    all(vapply(c("shiny", "bslib", "DT", "LLMR.shiny"),
               requireNamespace, logical(1), quietly = TRUE))) {
  run_focus_studio()
}
```

Index