Package 'LLMRagent'

July 27, 2026

Type Package

Title Reproducible Language-Model Agents for Research

Version 0.8.1

Description Large language model agents as governed research instruments, built on 'LLMR'. The package supports designed conversations and factorial experiments with declared tools and budgets. Each run produces an inspectable record that can be archived and checked.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.2)

Imports LLMR (>= 0.8.11), R6, tibble, rlang, cli, digest, jsonlite, httr2, utils

Suggests testthat (>= 3.0.0), knitr, rmarkdown, future, future.apply, dplyr, withr, callr, jsonvalidate, htmltools

Config/testthat/edition 3

VignetteBuilder knitr

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

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

NeedsCompilation no

Author Ali Sanaei [aut, cre]

Maintainer Ali Sanaei <sanaei@uchicago.edu>

Repository CRAN

Date/Publication 2026-07-27 18:10:09 UTC

Contents

---

add_edge *Add an edge to a workflow*

Description

Connects two nodes. With `when = NULL` the edge is unconditional; with a predicate `when = function(state) -> logical` it is taken only when the predicate holds. After a node, the runtime takes the first outgoing edge whose predicate is `TRUE` (or the unconditional edge). A back-edge forms a loop, bounded by `max_steps` in `run_workflow()`.

Usage

```
add_edge(wf, from, to, when = NULL)
```

Arguments

`wf`	An `agent_workflow`.
`from`, `to`	Node names.
`when`	Optional `function(state) -> logical`.

Value

The workflow, with the edge added.

See Also

`add_node()`, `run_workflow()`

---

add_node *Add a node to a workflow*

Description

A node transforms the shared `state`. It may be: an Agent (its `reply()` is called on `state[[input_key]]` and the result written to `state[[output_key]]`); a plain `function(state)` returning the new state; an evaluator (a function or `Agent + schema` writing a structured verdict into state, used by conditional edges); or a `human_gate()` (pauses the run for sign-off).

Usage

```
add_node(
  wf,
  name,
  node,
  input_key = "input",
  output_key = NULL,
  schema = NULL,
  ...
)
```

Arguments

wf	An agent_workflow.
name	Node name (unique within the workflow).
node	An Agent, a function(state), or a human_gate() marker.
input_key, output_key	For an agent node: where to read the prompt and write the reply in state (default "input" / the node name).
schema	For an evaluator agent node: a JSON schema; the parsed verdict is written to state[[output_key]].
...	Reserved.

Value

The workflow, with the node added. The first node added is the entry.

See Also

agent_workflow(), add_edge()

---

agent *Create an agent*

Description

An agent is a persona plus a model: it remembers its conversation (see memory), can call R functions you expose as tools (via LLMR::llm_tool()), and refuses further calls when its budget() runs out. Use agent$chat() for a stateful conversation, agent$ask_structured() for schema-shaped answers, and pass agents to conversation(), debate(), focus_group(), interview(), or deliberate() for multi-agent work.

Usage

```
agent(
  name,
  config,
  persona = NULL,
  tools = list(),
  memory = memory_buffer(),
  budget = LLMRagent::budget(),
  guardrails = NULL,
  quiet = FALSE
)
```

Arguments

name	Display name (used in transcripts).
config	An LLMR::llm_config() for a generative model.
persona	Optional system prompt: who this agent is, what it wants, how it speaks. For social-science personas, write it like a character brief: background, dispositions, speech style.
tools	A LLMR::llm_tool() or list of them. Tool calls the model makes are executed automatically and fed back until it answers.
memory	A memory object; default keeps the last 40 messages.
budget	A budget(); default unlimited.
guardrails	Optional guardrails() to check the agent's inputs, outputs, and tool calls. A blocked check raises llmragent_guardrail_block and is recorded as an event; default NULL means no guardrails.
quiet	If TRUE, chat() does not echo replies to the console.

Details

Two design decisions worth knowing:

- **Failures are errors, not replies.** If a call fails, the typed LLMR condition propagates; nothing is written into memory, so an API hiccup is never stored as something the model said.
- **Budgets are checked at every call boundary.** Call and tool-call limits stop the next round. The token limit stops the next round once recorded use reaches it; one response can cross that threshold.

Value

An Agent (R6) object.

See Also

budget(), memory, agent_as_tool(), agent_pipeline(), conversation(), agent_experiment()

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.7)
ada <- agent("Ada", cfg,
             persona = "You are Ada, a meticulous statistician. Be brief.")
ada$chat("In one sentence: what is overfitting?")
ada$chat("And how would you detect it?")   # remembers the thread
ada$chat("Walk me through cross-validation.", stream = TRUE)  # live tokens
ada$usage()

## End(Not run)
```

---

agent_as_tool *Expose an agent as a tool for other agents*

Description

Turns an Agent into an `LLMR::llm_tool()`, so other agents can delegate to it. A supervisor with specialist sub-agents in its `tools` decides for itself when to consult whom; the specialist's reply comes back as the tool result, and the supervisor continues with it.

Usage

```
agent_as_tool(x, name = NULL, description = NULL)
```

Arguments

`x`	The Agent to expose.
`name`	Tool name shown to the calling model. Default `ask_<name>`, lower-cased and sanitized.
`description`	What the calling model is told about this specialist. Defaults to the first sentence of the persona, prefixed by the agent's name. Write this the way you would brief a colleague: what the specialist knows and when to consult it.

Details

Three properties make delegation safe and auditable:

- **Spend is attributed.** A consultation runs through the specialist's own machinery, so its `usage()` and `trace()` record the work it did, while the supervisor's trace records the tool call.
- **Budgets nest.** Give the specialist its own `budget()`; when it is exhausted, the supervisor receives the budget error as the tool result (an `"ERROR: ..."` string) and can carry on without it.
- **Consultations are stateless.** Each delegated question goes through `reply()`: the specialist's persona applies, but nothing is written to its memory, so concurrent supervisors cannot contaminate each other.

Value

An `LLMR::llm_tool()` object, ready for `agent(tools = ...)`.

See Also

`agent()`, `agent_pipeline()`, `agent_fanout_synthesis()`

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.3)

statistician <- agent("Stat", cfg,
  persona = "A PhD statistician. Precise about assumptions and pitfalls.",
  budget  = budget(max_calls = 5))   # the specialist has its own ceiling
historian <- agent("Hist", cfg,
  persona = "An economic historian. Strong on institutional context.")

supervisor <- agent("Lead", cfg,
  persona = "A research lead. Consult your specialists, then synthesize.",
  tools = list(agent_as_tool(statistician), agent_as_tool(historian)))

supervisor$chat(
  "We observe falling crime and rising policing budgets across cities.
   What would it take to argue causality here?")
statistician$usage()   # the consultation is on the specialist's meter

## End(Not run)
```

`agent_experiment` *Run a factorial agent experiment*

Description

Takes a design frame (one row per condition), runs `run_fn` once per condition and replication, and returns the design with `rep`, `result` (list-column), `error`, and `duration` columns. A failing cell records its error message and does not stop the others; re-run failures by filtering on `!is.na(error)`.

Usage

```
agent_experiment(design, run_fn, reps = 1L, parallel = FALSE, quiet = FALSE)

## S3 method for class 'agent_experiment'
print(x, ...)
```

Arguments

design	A data frame; each row is a condition, each column a factor of the design (personas, prompts, treatments, model names, ...).
run_fn	function(cond, rep) where cond is one row of design as a named list. Whatever it returns is stored in the result list-column.
reps	Replications per condition (default 1).
parallel	If TRUE, cells run concurrently via the future framework (set a plan first, e.g. future::plan(future::multisession)).
quiet	FALSE prints one progress line per completed cell.
x	An agent_experiment object.
...	Ignored.

Details

run_fn receives the condition as a named list plus the replication number, and should build its agents *inside* the function so every cell starts fresh:

```
run_fn <- function(cond, rep) {
  cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.9)
  panel <- list(
    agent("A", cfg, persona = cond$persona_a),
    agent("B", cfg, persona = cond$persona_b)
  )
  deliberate(panel, cond$proposal, quiet = TRUE)
}
```

A note on seeds: model replies are sampled server-side, so a local seed never affects them. Set one only when your code draws random numbers itself (a run_fn that randomizes stimuli, or the "random" turn policy inside it); under parallel = TRUE the workers then use statistically sound parallel streams (future.seed). Combine with LLMR::llm_log_enable() to keep a per-call audit file of the whole experiment.

Value

design expanded by replication, plus rep, result, error, and duration columns.

See Also

deliberate(), debate(), LLMR::llm_usage()

Examples

```
## Not run:
design <- expand.grid(
  temperature = c(0.2, 1.0),
  framing = c("gains", "losses"),
  stringsAsFactors = FALSE
```

```
)
res <- agent_experiment(design, reps = 3, run_fn = function(cond, rep) {
  cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b",
                          temperature = cond$temperature)
  a <- agent("Subject", cfg, quiet = TRUE)
  a$reply(paste("Decide under", cond$framing, "framing: ..."))
})

## End(Not run)
```

`agent_fanout_synthesis` *Work a hard problem with one strong model and many cheap ones*

Description

A four-stage orchestration:

Usage

```
agent_fanout_synthesis(
  problem,
  strong_config,
  cheap_config,
  n_approaches = 4L,
  verify = TRUE,
  quiet = FALSE,
  ...
)
```

Arguments

`problem`	The problem statement (character scalar). Be complete: workers see nothing but this and their assigned approach.
`strong_config`	An `LLMR::llm_config()` for the strong (planner / synthesizer / verifier) model.
`cheap_config`	An `LLMR::llm_config()` for the cheap worker model.
`n_approaches`	Number of independent lines of attack (default 4).
`verify`	If TRUE (default), run the verification stage and one revision round when the verifier finds a substantive flaw.
`quiet`	FALSE prints stage progress.
`...`	Passed to `LLMR::call_llm_par()` for the worker stage (e.g. `tries`, `progress`).

Details

1. **Plan** (strong model, 1 call): decompose the problem into `n_approaches` genuinely different lines of attack.
2. **Work** (cheap model, `n_approaches` parallel calls): each line is pursued independently, blind to the others.
3. **Synthesize** (strong model, 1 call): weigh the drafts against one another, resolve contradictions, and write the answer.
4. **Verify** (strong model, 1 call, optional): attack the synthesized answer; if a real flaw is found, one revision pass runs.

The economics: two to three strong-model calls regardless of how wide the fan-out is, with the bulk of tokens billed at the cheap model's rate. The returned object keeps every intermediate product, so you can audit what each worker contributed and what the verifier objected to.

Value

A list of class `agent_fanout_result`:

`answer` The final answer (character).

`plan` Tibble of approaches: `title`, `instructions`.

`workers` Tibble: `approach`, `output`, `success`, plus token diagnostics from `LLMR::call_llm_par()`.

`verification` The verifier's structured critique (or NULL).

`revised` TRUE when the revision pass ran.

Examples

```
## Not run:
strong <- LLMR::llm_config("deepseek", "deepseek-reasoner")
cheap  <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.8)
out <- agent_fanout_synthesis(
  "A city of 1M residents wants to halve traffic deaths in 5 years
   with a budget of $40M. Propose the most cost-effective portfolio
   of interventions, with rough numbers.",
  strong_config = strong, cheap_config = cheap, n_approaches = 5
)
cat(out$answer)
out$workers[, c("approach", "success")]

## End(Not run)
```

---

agent_manifest *Build the study manifest for a run*

Description

One object tying together the recorded design, personas, declared tool specifications, orchestration metadata, served model identifiers, generation parameters, and package versions. Its `manifest_hash` changes when a hashed component changes. It does not hash transcripts, replies, or unrecorded ambient state such as values captured by a tool closure.

Usage

```
agent_manifest(run)
```

Arguments

run	An object accepted by as_agent_run() (a chat agent, a conversation, a preset, a pipeline, an experiment, an agent_fanout_result, or an agent_run).

Value

An object of class agent_manifest (a list); see Details. Print shows the short hash, kind, models, and headline counts.

See Also

hash_persona(), hash_tool_spec(), archive_agent_study(), LLMR::llm_hash()

---

agent_pipeline *Run input through a chain of agents*

Description

Each agent receives the previous agent's output as its message (the first receives input), transforms it according to its persona, and hands the result on. A common use is a fixed sequence of narrow specialists: extract, then translate, then critique. Each stage is easy to inspect, test, and swap.

Usage

```
agent_pipeline(agents, input, quiet = FALSE, ...)
```

Arguments

agents	A list of Agents, in order. A single agent is allowed.
input	The text handed to the first agent.
quiet	If FALSE (default), each stage's output prints as it arrives.
...	Passed to each agent's underlying LLMR call.

Details

Stages are stateless (reply()), so the same agents can serve in several pipelines, and a pipeline can run many inputs without cross-contamination. Every intermediate product is kept: the returned steps tibble has one row per stage with the exact input and output of each agent.

Value

An object of class agent_pipeline_run: a list with steps (tibble: step, agent, input, output) and output (the final text). as.data.frame() returns the steps.

See Also

agent_as_tool() for model-directed (rather than fixed) routing.

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.3)

run <- agent_pipeline(
  list(
    agent("Extractor", cfg, persona =
      "Extract every factual claim as a numbered list. Nothing else."),
    agent("Checker", cfg, persona =
      "For each numbered claim, mark VERIFIABLE or VAGUE, one line each."),
    agent("Editor", cfg, persona =
      "Rewrite the original message keeping only VERIFIABLE claims.")
  ),
  input = "Our app doubled retention, won three awards, and users love it."
)
run$output      # the final text
run$steps       # every intermediate product

## End(Not run)
```

---

agent_robustness *Run a robustness battery*

Description

Runs a procedure across the cross-product of perturbation axes and reports, by axis, how much the result moves. The procedure is your run_fn; the perturbations reach it through an optional third argument, perturb, so you write the procedure once and the battery varies its inputs.

Usage

```
agent_robustness(
  run_fn,
  design = NULL,
  reps = 1L,
  vary = list(),
  measure = NULL,
  baseline = "first",
  config = NULL,
  parallel = FALSE,
  quiet = TRUE,
  ...
)
```

Arguments

run_fn	The procedure, function(cond, rep[, perturb]), returning a result whose stability is assessed via measure.
design	Optional baseline conditions (a data frame); one empty condition if NULL.
reps	Replications per cell.
vary	A named list of axes: bare level vectors or axis specs from vary_models() and related helpers.
measure	A function result -> scalar (or a field name) producing the quantity whose stability is assessed. If NULL, the result is used when it is already scalar.
baseline	Which level of each axis is the reference ("first").
config	A base config (used to build perturb$config).
parallel	Passed to agent_experiment().
quiet	Passed to agent_experiment().
...	Passed to agent_experiment().

Details

run_fn may be function(cond, rep) (the existing agent_experiment() contract) or function(cond, rep, perturb). perturb is a list with config (the base config with this cell's model/temperature applied), persona(x) (apply this cell's persona variant to a base persona), prompt(x) (apply this cell's prompt variant), and reorder(options) (apply this cell's option permutation). A two-argument run_fn still runs; then only the model/temperature axes affect the run.

Value

An object of class agent_robustness: a list with cells (the full perturbed design with measure_value), by_axis (one row per axis-level with instability, dispersion, agreement_alpha, failure_rate, flips_vs_baseline, delta_mean), and overall (a fragile flag).

See Also

agent_experiment(), LLMR::llm_replicate(), LLMR::llm_agreement()

Examples

```
## Not run:
batt <- agent_robustness(
  run_fn = function(cond, rep, perturb) {
    a <- agent("S", perturb$config, persona = perturb$persona("A cautious voter."))
    a$reply(perturb$prompt("Do you support the policy? yes/no."))
  },
  vary = list(temperature = c(0, 1), model = c("openai/gpt-oss-20b")),
  measure = function(r) tolower(trimws(r))
)
batt$by_axis

## End(Not run)
```

agent_tool *Define a governed tool*

Description

Like LLMR::llm_tool(), but the tool also declares how it behaves as a research instrument: whether it reads, writes, or reaches outside the session; whether a human must approve each call; and hard per-tool limits on the number of calls, wall-clock time, and result size. The returned object is an ordinary llmr_tool, so it passes to agent() and the tool loop exactly as a plain tool does; the governance is carried alongside and enforced on every call.

Usage

```
agent_tool(
  fn,
  name,
  description,
  parameters = NULL,
  required = NULL,
  side_effects = c("read", "write", "external", "none"),
  requires_approval = FALSE,
  timeout_s = NULL,
  max_calls = Inf,
  max_bytes = Inf
)
```

Arguments

fn	The R function to expose. Called with the model's arguments by name, exactly as in LLMR::llm_tool().
name	Tool name shown to the model.
description	One or two sentences for the model.
parameters	A named list of JSON-Schema properties, or a full schema object (as in LLMR::llm_tool()).
required	Character vector of required argument names.
side_effects	What the tool does in the world: "read" (default), "write", "external" (reaches a network or service), or "none".
requires_approval	If TRUE, each call pauses for human sign-off (see human_gate()); the agent then runs a controllable tool loop so the pause is possible. Default FALSE.
timeout_s	Optional wall-clock limit per call (seconds). Timed calls run in a callr child process, which is terminated at the limit.
max_calls	Maximum times this **tool object** may run (Inf by default). A call beyond the limit is refused (the model is told, not the tool executed). The counter lives in the tool object, so it is shared if the same agent_tool() object is reused across agents or experiment cells; build a fresh tool per independent cell (as you would a fresh agent) when each cell should get its own budget.

`max_bytes` Maximum result size in bytes (as measured by `nchar(type = "bytes")`); a larger result is truncated at a character boundary within the cap and flagged.

Details

A tool's policy is folded into the study manifest via `hash_tool_spec()`, so tightening a limit (a different apparatus) changes the manifest hash. Each invocation is recorded with the hashes of its arguments and result, its status, and its duration, surfaced at the `"tool"` level of `as_agent_run()`.

Value

An object of class `agent_tool` and `llmr_tool`, with its governance policy in the ordinary `governance` field.

See Also

`LLMR::llm_tool()`, `hash_tool_spec()`, `guardrail()`, `human_gate()`

Examples

```
lookup <- agent_tool(
  function(city) paste0("22C in ", city),
  name = "get_weather", description = "Current weather for a city.",
  parameters = list(city = list(type = "string")),
  side_effects = "external", max_calls = 5
)
```

---

agent_workflow *Build an agent workflow (a small, explicit graph)*

Description

A workflow is a directed graph whose nodes transform a shared, explicit `state` list. Build it with `agent_workflow()` then `add_node()` and `add_edge()`; run it with `run_workflow()`. The runtime is minimal: sequential execution, one mutable state, and a checkpoint after each node. A run is therefore auditable and resumable. Reach for it only when a study genuinely needs branching, looping, or mid-run human review; `agent_pipeline()` and `conversation()` remain the simpler interfaces.

Usage

```
agent_workflow(name)
```

Arguments

`name` A label for the workflow.

Value

An agent_workflow object (built up immutably by add_node() / add_edge()).

See Also

add_node(), add_edge(), run_workflow(), resume_workflow(), fork_workflow(), replay_run()

Examples

```
wf <- agent_workflow("classify") |>
  add_node("clean", function(state) { state$x <- trimws(state$input); state }) |>
  add_node("label", function(state) { state$label <- nchar(state$x) > 3; state }) |>
  add_edge("clean", "label")
run <- run_workflow(wf, input = "  hello  ")
run$state$label
```

---

approve_tool_call *Approve, reject, or edit a pending tool call*

Description

When a run pauses at an approval-gated tool, it surfaces a checkpoint (the checkpoint field of the llmragent_pending_approval condition, or the object returned by human_gate() in batch mode). Inspect checkpoint$pending (the tool name, arguments, and an argument hash), then record a decision with this function and continue with resume_run().

Usage

```
approve_tool_call(
  checkpoint,
  decision = c("approve", "reject", "edit"),
  edit = NULL
)
```

Arguments

checkpoint	A llmragent_checkpoint.
decision	"approve" (run the tool as requested), "reject" (skip it; the model is told the call was denied), or "edit" (run it with edit-modified arguments).
edit	For decision = "edit", the replacement argument list.

Value

The checkpoint with the decision recorded.

See Also

human_gate(), resume_run(), agent_tool()

`archive_agent_study` *Seal an agent study to a directory*

Description

Writes a self-contained, hash-sealed archive of a run: the study `agent_manifest()` (`manifest.json`), the transcript (`transcript.csv`), the event / call / tool / state views, the per-call provenance (`calls.jsonl`, the LLMR audit log copied verbatim when one was kept), any artifacts, a drafted methods note (`README-methods.md`), and a `hashes.sha256` manifest over every file written. The archive is the supplementary material a paper can ship: it carries the declared specification and the calls' provenance, not just the prose.

Usage

```
archive_agent_study(
  run,
  path,
  include_messages = TRUE,
  redact = NULL,
  formats = c("csv", "jsonl", "rds"),
  overwrite = FALSE
)
```

Arguments

`run`	An object accepted by `as_agent_run()` (an Agent, a conversation or preset result, a pipeline, an experiment, or an `agent_run`).
`path`	Directory to write into; created (recursively) if absent.
`include_messages`	If `TRUE` (default), write the transcript and the free-text columns of the tool and state tables. If `FALSE`, those tables are still written but their free-text columns are blanked, keeping only structure, hashes, and metadata; the records in `calls.jsonl` likewise omit the request body and reply text (each record keeps its precomputed `request_hash`, so the join invariant holds).
`redact`	Optional redaction applied to the free-text columns of the written transcript, tool, and state tables (`text`, `content`, `arguments`, `result`, `response_text`) and to the reply text of the records in `calls.jsonl` – never to hashes, and never to a request body, which is the call's identity (omit request bodies with `include_messages = FALSE`). Either a `function(text) -> text`, or a character vector of regular expressions, each of which is replaced by `"[REDACTED]"`. Redaction is applied to the on-disk copy after hashing, so the join invariant holds.
`formats`	Which optional formats to write, any of `"csv"` (the tabular views), `"jsonl"` (events and calls; always written), and `"rds"` (the privacy-filtered, data-only run snapshot as `run.rds`). Defaults to all three.
`overwrite`	If `FALSE` (default), refuse a non-empty `path`. Set to `TRUE` to replace archive files in an existing directory.

Details

The archive is privacy-preserving at its serialization boundary: `run.rds` is a data-only snapshot of the same projected levels written to the text formats. Live agents, callers, tool functions, model configurations, and configuration secrets are never serialized into the archive. Message omission and redaction apply to this snapshot as well as the text files.

Hashes are identity, not outcome. The `request_hash` in `calls.jsonl` is computed over the original request, so a call read back from the archive still matches the same call issued live. Redaction therefore never touches a hash or a request body, and when an LLMR audit log is present it is copied byte-for-byte rather than reserialized – unless a privacy lever is engaged: with `include_messages = FALSE` each copied record's request body and reply text are removed (its precomputed `request_hash` is kept, so the join survives), and with `redact` the copied records' reply text is scrubbed.

Value

Invisibly, an object of class `agent_archive`: a list with `path`, `files` (relative paths written), `manifest_hash`, and `n_calls`. Print lists what was written and confirms the seal.

See Also

`agent_manifest()`, `as_agent_run()`, `LLMR::llm_log_read()`

Examples

```
## Not run:
a <- agent("Aria", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
a$chat("Hello")
archive_agent_study(a, tempfile("study_"))

## End(Not run)
```

---

as_agent_run *Convert an LLMRagent result to a unified run object*

Description

`as_agent_run()` turns any high-level result into an `agent_run`: a single object that exposes a run at five levels through one tidy accessor, `tibble::as_tibble(run, level = c("utterance","event","call","tool","sta`
and that backs `agent_manifest()`, `archive_agent_study()`, `diagnostics()`, and `report()`.

Usage

```
as_agent_run(x, ...)

## S3 method for class 'agent_run'
as_tibble(x, ..., level = c("utterance", "event", "call", "tool", "state"))
```

Arguments

x	An Agent; a conversation, debate, focus group, interview, or deliberation; an agent_pipeline_run; an agent_fanout_result; an agent_experiment; or a LLMR::call_llm_par()-style result frame.
...	Unused.
level	The grain to return: "utterance" (analysis grain), "event" (every span), "call" (one canonical LLMR::llm_response_record() row per model call), "tool" (tool invocations with arg/result hashes), or "state" (each participating agent's memory at run end).

Details

Provenance is captured during every model call, so converting costs nothing during the run; the views are built on demand. For a bare Agent, the result is a live view of the agent's session so far.

Value

An object of class agent_run.

See Also

agent_manifest(), archive_agent_study(), diagnostics(), report()

---

budget *Spending and effort limits for an agent*

Description

Call, tool-call, and elapsed-time limits are checked before every model round. When one is exhausted, the agent raises a condition of class llmragent_budget_error without making the next call. max_tokens is a recorded-use gate: the agent stops before the next call once recorded usage reaches the limit, but one response can take the total past it because the response's token count is not known in advance.

Usage

```
budget(
  max_calls = Inf,
  max_tokens = Inf,
  max_tool_calls = Inf,
  max_seconds = Inf
)
```

Arguments

max_calls	Maximum number of model calls, counting chat exchanges, memory compactions, and retrieval-memory embedding operations.
max_tokens	Stop before the next model call once recorded sent and received tokens reach this value.
max_tool_calls	Maximum executed tool invocations.
max_seconds	Wall-clock ceiling, measured from the agent's first call.

Value

An object of class agent_budget.

Examples

```
b <- budget(max_calls = 10, max_tokens = 50000)

## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")
frugal <- agent("Frugal", cfg, budget = budget(max_calls = 2))
frugal$chat("one")
frugal$chat("two")
tryCatch(frugal$chat("three"),
         llmragent_budget_error = function(e) "refused before spending")

## End(Not run)
```

---

check_state_leakage *Detect shared state across experiment cells*

Description

check_state_leakage() inspects the cells of an agent_experiment() (or a plain list of convertible results) and reports whether any cell shares a live agent with another. A correct experiment builds its agents *inside* run_fn, so each cell gets fresh agents with distinct ids; an agent built once *outside* run_fn and reused leaks its memory and identity across cells, which silently couples conditions that should be independent.

Usage

```
check_state_leakage(x)
```

Arguments

x	An agent_experiment() result (the tibble with a result list-column), or a plain list whose elements are Agents or agent_run objects (each element treated as one cell).

Details

The check is read-only and conservative: every cell is converted through `as_agent_run()` inside a `tryCatch()`, and a cell whose result is not run-able (a number, a string, anything off the provenance path) is skipped rather than treated as evidence. Two kinds of leak are reported:

- `shared_agent_instance`: one `agent_id` participates in two different cells. Distinct cells with a fresh agent each cannot collide on an id, so a shared id is direct evidence of one reused instance.
- `memory_bleed`: a stronger signal on top of a shared id, raised when the later cell's agent memory already contains content hashed from the earlier cell, i.e. the agent literally remembers the prior condition.

Value

An object of class `agent_leakage_report`: a list with `leaks` (a tibble with columns `cell_i`, `cell_j`, `agent_id`, `kind`, `evidence`), `clean` (`TRUE` when no leaks were found), and `n_cells`.

See Also

`agent_experiment()`, `as_agent_run()`

Examples

```
## Not run:
design <- data.frame(framing = c("gains", "losses"))

# leaks: one agent is built once and reused by every cell
shared <- agent("S", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
bad <- agent_experiment(design, reps = 1, run_fn = function(cond, rep) {
  shared$chat(cond$framing); shared
})
check_state_leakage(bad)          # clean = FALSE

# clean: a fresh agent is built inside every cell
good <- agent_experiment(design, reps = 1, run_fn = function(cond, rep) {
  a <- agent("S", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
  a$chat(cond$framing); a
})
check_state_leakage(good)         # clean = TRUE

## End(Not run)
```

`conversation` *Run a multi-agent conversation*

Description

Agents talk over a shared transcript. At each turn the next speaker (chosen by `turn_policy`) receives the full dialogue so far, attributed by name, plus an instruction to answer in character; the reply is appended and the next turn begins. The conversation ends after `max_turns` utterances or when `stop_when(transcript)` returns `TRUE`.

Usage

```
conversation(
  agents,
  topic,
  opening = NULL,
  opening_by = "Facilitator",
  turn_policy = c("round_robin", "random", "moderator"),
  moderator = NULL,
  max_turns = 2L * length(agents),
  stop_when = NULL,
  instruction = NULL,
  msg_mode = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

`agents`	A list of Agent objects (names must be unique).
`topic`	What the conversation is about; included in every speaker's instructions.
`opening`	Optional opening statement placed on the transcript before the first turn (attributed to `opening_by`).
`opening_by`	Name to attribute the opening to. Default "Facilitator".
`turn_policy`	One of `"round_robin"`, `"random"`, `"moderator"`.
`moderator`	An Agent; required for the moderator policy.
`max_turns`	Total number of utterances to collect.
`stop_when`	Optional `function(transcript_tibble) -> logical`; checked after every utterance.
`instruction`	Extra instruction appended to every speaker's system message (e.g. "Answer in at most three sentences.").
`msg_mode`	Message construction for this run: `"roleflip"` (default) or `"flat"`. `NULL` (the default) uses `getOption("LLMRagent.msg_mode")`, else `"roleflip"`. An explicit value overrides the global option for this call only. See the Message construction section.
`quiet`	If FALSE (default), utterances print as they arrive.
`...`	Passed to each agent's underlying LLMR call.

Details

Turn policies:

- `"round_robin"`: agents speak in the order given, repeatedly.
- `"random"`: a random speaker each turn, never the same agent twice in a row. Set a seed first (e.g. `set.seed(110)`) for a reproducible order.
- `"moderator"`: after each utterance the `moderator` agent chooses who speaks next (a structured one-token decision), which lets the moderator shape the turn order at the cost of one extra model call per turn.

Replies are stateless (Agent's `reply()`): the shared transcript is the single source of truth, so the same agents can be reused across conversations without cross-contamination.

Value

An object of class `agent_conversation`: a list with `transcript` (tibble: `turn`, `speaker`, `text`), `topic`, and `agents` (names).

Message construction

By default each speaker sees the conversation role-flipped: its own prior turns are `assistant` messages and every other speaker's are labeled `user` messages (via `LLMR::transcript_as_messages()`), which marks what the model already said and reduces self-repetition. `msg_mode = "flat"` (or `options(LLMRagent.msg_mode = "flat")`) reverts to the legacy construction that pastes the whole attributed transcript into one `user` message – useful for reproducing pre-0.7.x runs. The same control applies to the presets `debate()`, `focus_group()`, `interview()`, and `deliberate()`.

See Also

`debate()`, `focus_group()`, `interview()`, `deliberate()`

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.8)
a <- agent("Rosa",  cfg, persona = "A pragmatic city planner.")
b <- agent("Hugo",  cfg, persona = "A skeptical economist.")
conv <- conversation(list(a, b),
                     topic = "Should the city pedestrianize its center?",
                     max_turns = 6)
conv$transcript

## End(Not run)
```

debate *Structured debate between two agents*

Description

Alternating statements in three phases: opening, rounds rebuttals each, and closing. An optional judge then delivers a structured verdict. The phase labels make it easy to analyze argument development over time.

Usage

```
debate(
  pro,
  con,
  topic,
  rounds = 2L,
  judge = NULL,
  msg_mode = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

pro, con	Agents arguing for and against.
topic	The motion being debated.
rounds	Number of rebuttal exchanges (default 2).
judge	Optional Agent; if supplied, returns a verdict with a winner, a confidence, and reasoning.
msg_mode	Message construction, "roleflip" (default) or "flat"; NULL uses getOption("LLMRagent.msg_mode") See conversation().
quiet	Passed through; FALSE prints utterances live.
...	Passed to the agents' underlying LLMR calls.

Value

An object of class agent_debate: a list with transcript (tibble: turn, phase, speaker, text), verdict (list or NULL), and motion. as.data.frame() returns the transcript.

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.7)
d <- debate(
  pro = agent("Pro", cfg, persona = "You argue FOR the motion, rigorously."),
  con = agent("Con", cfg, persona = "You argue AGAINST the motion, rigorously."),
```

```
  topic = "Social media does more harm than good to democratic discourse.",
  judge = agent("Judge", cfg, persona = "A strict, impartial debate judge.")
)
d$verdict

## End(Not run)
```

`deliberate` *Group deliberation with a recorded vote*

Description

Agents discuss a proposal for a fixed number of rounds (everyone speaks each round, seeing the discussion so far), then vote independently and privately through structured output. The tidy return supports comparing votes cast after deliberation with positions voiced during it.

Usage

```
deliberate(
  agents,
  proposal,
  rounds = 2L,
  options = c("yes", "no", "abstain"),
  msg_mode = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

`agents`	A list of Agents.
`proposal`	The proposal under deliberation (character scalar).
`rounds`	Discussion rounds before the vote (default 2). `rounds = 0` skips the discussion: the panel votes on the bare proposal.
`options`	Vote options. Default `c("yes", "no", "abstain")`.
`msg_mode`	Message construction, `"roleflip"` (default) or `"flat"`; `NULL` uses `getOption("LLMRagent.msg_mode")`. See `conversation()`.
`quiet`	FALSE prints the deliberation live.
`...`	Passed to the underlying LLMR calls.

Value

An object of class `agent_deliberation`: a list with `transcript` (tibble: `turn`, `round`, `speaker`, `text`), `votes` (tibble: `voter`, `vote`, `reason`), `tally` (table), `decision` (modal vote, NA on ties), and `proposal`. `as.data.frame()` returns the transcript.

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.9)
panel <- list(
  agent("Aila", cfg, persona = "Data-driven, cautious about side effects."),
  agent("Bo",   cfg, persona = "Mission-driven, impatient with delay."),
  agent("Cyn",  cfg, persona = "A budget hawk.")
)
d <- deliberate(panel, "Adopt a four-day work week for one pilot year.")
d$tally; d$decision

## End(Not run)
```

---

diagnostics.agent_experiment

Machine-readable diagnostics for an agent experiment

Description

Returns the cell-level health numbers behind an `agent_experiment()` result: how many cells ran, how many failed, the failure rate, and the total wall-clock seconds. When the frame carries a `rep` column, the number of distinct conditions and the replication count are reported too. Robust to a frame missing the `error`, `duration`, or `rep` columns.

Usage

```
## S3 method for class 'agent_experiment'
diagnostics(x, ...)
```

Arguments

`x`	An `agent_experiment()` result (a tibble of class `agent_experiment`).
`...`	Unused.

Value

A one-row tibble with `n_cells`, `n_failed`, `n_ok`, `failure_rate`, `total_duration_s`, and (when a `rep` column is present) `n_conditions` and `reps`.

See Also

`agent_experiment()`, `report()`

diagnostics.agent_run *Machine-readable diagnostics for an agent run*

Description

Returns the small set of health numbers behind an agent_run: call counts, token totals, tool and blocked-event counts, wall-clock duration, and the study's `manifest_hash`. Token and outcome figures are read through `LLMR::llm_usage()` over the run's call-level rows, so they match the rest of the LLMR ecosystem to the integer.

Usage

```
## S3 method for class 'agent_run'
diagnostics(x, ...)

## S3 method for class 'Agent'
diagnostics(x, ...)
```

Arguments

`x`	An object accepted by `as_agent_run()` (an Agent, a conversation or preset result, a pipeline, an experiment, or an `agent_run`).
`...`	Unused.

Value

A one-row tibble with `run_id`, `kind`, `n_calls`, `n_ok`, `n_failed`, `ok_rate`, `n_truncated`, `n_filtered`, `tokens_sent`, `tokens_received`, `tokens_total`, `reasoning_tokens`, `n_tool_calls`, `n_blocked`, `duration_s`, and `manifest_hash`.

See Also

`report()`, `LLMR::llm_usage()`, `agent_manifest()`

Examples

```
## Not run:
a <- agent("Aria", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
a$chat("Hello")
diagnostics(a)

## End(Not run)
```

diagnostics.persona_audit *Machine-readable diagnostics for a persona audit*

Description

A one-row summary of a `persona_audit()`: the number of personas, how many tripped the lexical scan, the worst single hit count, and the mean model caricature score (`NaN` when no model layer ran).

Usage

```
## S3 method for class 'persona_audit'
diagnostics(x, ...)
```

Arguments

`x`	A `persona_audit()` result.
`...`	Unused.

Value

A one-row tibble with `n_personas`, `n_flagged`, `max_hits`, and `mean_caricature`.

See Also

`persona_audit()`

focus_group *A moderated focus group*

Description

The moderator puts each question to the group; every participant answers (speaking order rotates across questions so nobody speaks first every round); participants see the discussion so far, as in a real group. The moderator closes with a synthesis of themes and disagreements.

Usage

```
focus_group(
  moderator,
  participants,
  topic,
  questions = NULL,
  n_questions = 3L,
```

```
  msg_mode = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

moderator	An Agent running the group.
participants	A list of Agents.
topic	The study topic (context for everyone).
questions	Character vector of questions. If NULL, the moderator drafts n_questions itself, which is useful for piloting.
n_questions	Number of questions to draft when questions is NULL.
msg_mode	Message construction, "roleflip" (default) or "flat"; NULL uses getOption("LLMRagent.msg_mode") See conversation().
quiet	FALSE prints the session live.
...	Passed to the underlying LLMR calls.

Value

An object of class agent_focus_group: a list with transcript (tibble: turn, question_id, speaker, text), questions, summary (the moderator's synthesis), and topic. as.data.frame() returns the transcript.

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.9)
fg <- focus_group(
  moderator = agent("Moderator", cfg, persona = "A neutral focus-group moderator."),
  participants = list(
    agent("Maya", cfg, persona = "A 34-year-old nurse, prudent with money."),
    agent("Tom",  cfg, persona = "A 22-year-old gig worker, risk-tolerant."),
    agent("Ines", cfg, persona = "A 58-year-old teacher nearing retirement.")
  ),
  topic = "Attitudes toward a 4-day work week",
  questions = c("What would a 4-day week change in your daily life?",
                "What worries you about it?")
)
fg$summary

## End(Not run)
```

`fork_workflow` *Fork a workflow run at a checkpoint*

Description

Copies the state at a chosen step into a new directory with a fresh run id, optionally mutating it, and runs forward from there. The original run is untouched: resume continues the same run, fork branches a new one.

Usage

```
fork_workflow(
  x,
  wf,
  at = NULL,
  new_dir = NULL,
  mutate = NULL,
  max_steps = 64L,
  quiet = TRUE,
  ...
)
```

Arguments

`x`	A `checkpoint_dir` or `agent_workflow_run`.
`wf`	The `agent_workflow` (needed to run the branch).
`at`	Step index to fork from (default: the last completed step).
`new_dir`	Directory for the branch (default: a fresh temp dir).
`mutate`	Optional `function(state) -> state` applied before running.
`max_steps, quiet, ...`	As in `run_workflow()`.

Value

An `agent_workflow_run` for the branch.

See Also

`run_workflow()`, `resume_workflow()`

guardrail *Define a guardrail*

Description

A guardrail is a named check run at one boundary of an agent: its input (the user text), its output (the model's reply), or a tool call (a tool's name and arguments before it runs, or its result after). The check returns TRUE to pass, or a short reason string to fail. On failure the guardrail either blocks (raises a typed condition and records the event), warns, or merely flags; in every case the decision is recorded as an event, so a blocked input is analyzable rather than invisible.

Usage

```
guardrail(
  name,
  check,
  on_fail = c("block", "warn", "flag"),
  stage = c("input", "output", "tool")
)
```

Arguments

name	A short label for the guardrail (appears in events).
check	A function `(payload, context) -> TRUE | reason`. `payload` is the text (input/output) or a `list(name=, arguments=, result=)` (tool). `context` is a small list with `stage`, `agent`, and `phase` (`"pre"` or `"post"` for tools). Return TRUE to pass, or a non-empty character reason to fail.
on_fail	What to do on failure: `"block"` (raise `llmragent_guardrail_block` and stop), `"warn"` (warn and continue), or `"flag"` (record only). Default `"block"`.
stage	Which boundary: `"input"`, `"output"`, or `"tool"`.

Value

An object of class `agent_guardrail`.

See Also

guardrails(), agent()

Examples

```
no_pii <- guardrail(
  "no_ssn",
  function(payload, context) {
    if (grepl("\\b\\d{3}-\\d{2}-\\d{4}\\b", payload)) "contains an SSN" else TRUE
  },
  stage = "input"
)
```

guardrails *Collect guardrails*

Description

Bundle one or more `guardrail()` objects to attach to an `agent()` via `agent(..., guardrails = guardrails(...))`.

Usage

```
guardrails(...)
```

Arguments

`...`	`guardrail()` objects.

Value

An object of class `agent_guardrails` (a list).

See Also

`guardrail()`

hash_persona *Hash a persona*

Description

A stable identity for a persona prompt, reusing LLMR's content-hash convention. Two agents with the same brief and name hash identically; any wording change flips the hash.

Usage

```
hash_persona(persona, name = NULL)
```

Arguments

`persona`	Character scalar (the persona brief) or a `persona_frame()`.
`name`	Optional agent name folded into the hash (two agents with the same brief but different display names are arguably different personas).

Value

A 64-character SHA-256 hex string (see `LLMR::llm_hash()`).

See Also

persona_frame(), agent_manifest()

---

hash_tool_spec *Hash a tool's declared specification*

Description

Hashes a tool's name, description, argument schema, governance policy (side effects, approval, limits), and R function body. Captured values in the function's enclosing environment are omitted, so this identifies the declared specification rather than the complete executable apparatus.

Usage

```
hash_tool_spec(tool)
```

Arguments

tool	An LLMR::llm_tool() or a governed agent_tool().

Value

A 64-character SHA-256 hex string.

See Also

agent_tool(), agent_manifest()

---

human_gate *Mark a point or a tool as requiring human approval*

Description

human_gate() is a marker used in two places. As a wrapper, human_gate(tool) returns the tool with its approval requirement set, equivalent to building it with agent_tool(..., requires_approval = TRUE). As a workflow node (Stage 4), it pauses a run for sign-off. In an agent's tool list, any approval-required tool makes the agent run a pausable tool loop.

Usage

```
human_gate(x, prompt = NULL)
```

Arguments

x	A tool (an llmr_tool / agent_tool()) to gate, or a name (label) when used as a standalone gate marker.
prompt	Optional text shown to the human reviewer.

Value

The gated tool (when x is a tool), or a gate marker object.

See Also

approve_tool_call(), resume_run(), agent_tool()

---

interview *A semi-structured interview*

Description

The interviewer works through a question list (or drafts one), asking one question at a time with an optional adaptive follow-up probing the respondent's previous answer. The returned object carries the tidy question/answer frame in $qa.

Usage

```
interview(
  interviewer,
  respondent,
  topic,
  questions = NULL,
  n_questions = 5L,
  follow_up = TRUE,
  msg_mode = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

interviewer, respondent	Agents.
topic	Interview topic.
questions	Character vector; if NULL the interviewer drafts n_questions.
n_questions	Number of questions to draft when questions is NULL.
follow_up	If TRUE (default), each scripted question may be followed by one adaptive probe based on the answer.
msg_mode	Message construction, "roleflip" (default) or "flat"; NULL uses getOption("LLMRagent.msg_mode") See conversation().
quiet	FALSE prints the exchange live.
...	Passed to the underlying LLMR calls.

Value

An object of class `agent_interview`: a list with `qa` (a tibble with `order`, `type`, `question`, and `answer`), `topic`, and provenance. `as.data.frame()` returns `qa`.

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b", temperature = 0.8)
iv <- interview(
  interviewer = agent("Interviewer", cfg,
                      persona = "A careful qualitative researcher."),
  respondent  = agent("Respondent", cfg,
                      persona = "A first-generation college student, reflective."),
  topic = "Experiences with online learning",
  n_questions = 3
)
iv

## End(Not run)
```

`llmragent-methods` *LLMR-family methods for LLMRagent run objects*

Description

LLMRagent registers methods on three generics LLMR defines: `LLMR::diagnostics()` (machine-readable health numbers), `LLMR::report()` (a methods-section draft), and `LLMR::reset()` (clear an apparatus's state). The token and outcome counts come from `LLMR::llm_usage()` over the run's call-level rows, so they agree with the rest of the ecosystem.

`llm_claim_lint` *Assert (or scope) prose against a run's claim type*

Description

Checks a piece of text for population-estimate language and either appends a scope sentence (`action = "scope"`, the default) or raises `llmragent_claim_error` (`action = "error"`). No current claim type authorizes population-estimate language. Used by `report()`; exported so custom report code can reuse it.

Usage

```
llm_claim_lint(text, run = NULL, action = c("scope", "error"))
```

Arguments

text	Character vector of prose to check.
run	An object accepted by as_agent_run() (supplies the claim type named in a scope message), or NULL.
action	"scope" (append a caveat when a population claim is found) or "error" (raise).

Value

The text (a character vector), possibly with a caveat appended.

See Also

mark_claim_type(), report()

---

load_agent *Load an agent from disk*

Description

Restores an agent saved with save_agent(): same persona, config, memory contents, budget, guardrails, and accounting. Because the config holds an environment-variable reference rather than a key, the loaded agent works immediately wherever that variable is set. Call, token, and tool counters carry over, so a budget keeps binding across sessions; the wall-clock (max_seconds) budget restarts at load.

Usage

```
load_agent(path, tools = list(), embed_config = NULL)
```

Arguments

path	File path written by save_agent().
tools	Tools to re-attach (functions are not serialized).
embed_config	Required only when the saved agent used memory_recall(); the embedding config to rebuild it with.

Value

An Agent.

mark_claim_type *Mark the kind of claim a run can support*

Description

Records, on a run, what its results may be used to argue. The three types are "instrument_pilot" (the run characterizes an instrument, not a population), "theory_probe" (it explores a mechanism or hypothesis), and "coding" (it annotates data, with reliability reported).

Usage

```
mark_claim_type(run, type = c("instrument_pilot", "theory_probe", "coding"))
```

Arguments

run	An object accepted by as_agent_run().
type	One of "instrument_pilot", "theory_probe", or "coding".

Details

The label flows into report(): prose that would overstate the claim is scoped or refused. None of these types turns model output into a population estimate.

Value

The run (an agent_run), invisibly, with the claim type recorded.

See Also

report()

Examples

```
## Not run:
run <- as_agent_run(my_deliberation)
run <- mark_claim_type(run, "theory_probe")
report(run)   # prose is scoped to a theory probe, not a population claim

## End(Not run)
```

`mcp_tools` *Expose MCP server tools to an agent, under governance*

Description

Connects to a Model Context Protocol server and returns its tools as `LLMR::llm_tool()` objects ready for `agent()`, wrapped so the agent's use of them is safe and recorded. The defaults are strict because MCP's documented attack surface (tool/schema poisoning, line jumping, rug pulls, confused-deputy) lives in exactly the trust an agent places in a server's tool descriptions.

Usage

```
mcp_tools(
  config,
  policy = c("read_only", "read_write"),
  approve_writes = TRUE,
  audit = TRUE,
  allow = NULL,
  pin_schemas = TRUE,
  transport = NULL,
  timeout_s = 30,
  max_bytes = 1e+06
)
```

Arguments

`config`	A connection spec: a list with `url` (HTTP) or `command` (stdio), or anything your `transport` understands.
`policy`	`"read_only"` (default) or `"read_write"`.
`approve_writes`	If `TRUE` (default), write/external calls pass a human gate.
`audit`	If `TRUE` (default), schemas/calls/results are recorded.
`allow`	Optional character vector of tool names to expose (others are dropped and logged).
`pin_schemas`	If `TRUE` (default), pin and re-check tool schemas.
`transport`	Test/extension seam: a function `(method, params) -> result` speaking JSON-RPC to the server. Default builds a real HTTP/stdio client.
`timeout_s, max_bytes`	Per-call limits.

Details

Defenses applied:

- **Read-only floor** (`policy = "read_only"`): the floor is an allowlist, not a denylist. A tool is exposed only when it is *positively* known to be read-only (its `annotations$readOnlyHint` is `TRUE`). Any tool that is not positively read-only (one with no annotations, or one that looks write-like) is refused unless `policy = "read_write"`. A malicious server cannot slip a writing tool past the floor by giving it a benign name and no annotations.

- **Human gate for writes** (`approve_writes = TRUE`): any write/external call pauses for sign-off (see `human_gate()`).
- **Schema pinning** (`pin_schemas = TRUE`): each tool's full advertised signature (input schema, description, and annotations) is hashed at first listing and re-verified before every call by re-listing the server's tools (`tools/list`; MCP has no per-tool fetch). A later change, or a server that refuses to let us re-verify (a re-listing that errors, drops the tool, or returns no schema), raises `llmragent_mcp_schema_drift` rather than trusting the new definition or failing open (the schema-drift defense). The re-check *fails closed*: if it cannot confirm the tool is unchanged, it refuses.
- **Description sanitation**: server descriptions are treated as untrusted, never spliced into a system prompt, and scanned for injection patterns; a flagged tool is downgraded to require approval (the line-jumping defense).
- **Audit** (`audit = TRUE`): tool schemas, call argument hashes, and result hashes are recorded.

Value

A list of `llmr_tool` objects, each carrying MCP governance metadata.

See Also

`agent()`, `agent_tool()`, `human_gate()`

Examples

```
## Not run:
# offline, via a fake transport returning canned JSON-RPC
fake <- function(method, params) {
  if (method == "tools/list") list(tools = list(list(
    name = "search", description = "Search docs.",
    inputSchema = list(type = "object",
                       properties = list(q = list(type = "string"))))))
  else list(content = list(list(type = "text", text = "result")))
}
tools <- mcp_tools(list(url = "http://localhost:9000"), transport = fake)

## End(Not run)
```

---

memory *Agent memory policies*

Description

An agent's memory decides which past messages enter the next context window. Three policies ship with the package; all are drop-in:

Usage

```
memory_buffer(keep = 40L)

memory_summary(threshold_chars = 12000L, keep_last = 10L, config = NULL)

memory_recall(embed_config, keep_recent = 6L, k = 4L)
```

Arguments

keep, keep_last, keep_recent, k, threshold_chars	Policy sizes; see above.
config	Optional LLMR::llm_config() used only for compaction summaries; NULL (default) summarizes with the agent's own model.
embed_config	An LLMR embedding config (e.g. llm_config("gemini", "gemini-embedding-001", embedding = TRUE)).

Details

- memory_buffer(keep): the last keep messages, verbatim. Simple, predictable, and right for most short-lived agents.
- memory_summary(threshold_chars, keep_last, config): unbounded conversations. When stored text exceeds the threshold, the agent automatically condenses everything but the most recent messages into one summary note before its next request, so context stays small without forgetting decisions. By default the agent's own model writes the summary; pass config to bill compaction to a cheaper model instead.
- memory_recall(embed_config, keep_recent, k): long-horizon recall. Older messages are embedded (via LLMR); at each turn the k most similar to the current input are injected alongside the recent tail. Attached to an agent, each embedding operation counts toward the agent's max_calls budget and appears in its trace as an embed event.

Value

A memory object to pass as agent(memory = ...).

Examples

```
m <- memory_buffer(keep = 10)
m$add("user", "hello")$add("assistant", "hi")
length(m$get())

## Not run:
cfg   <- LLMR::llm_config("groq", "openai/gpt-oss-20b")
cheap <- LLMR::llm_config("groq", "llama-3.1-8b-instant")

# an agent that summarizes its own past with the cheap model
scribe <- agent("Scribe", cfg,
                memory = memory_summary(threshold_chars = 8000,
                                        config = cheap))
```

```
# an agent that recalls relevant old exchanges by embedding similarity
emb <- LLMR::llm_config("gemini", "gemini-embedding-001", embedding = TRUE)
sage <- agent("Sage", cfg, memory = memory_recall(emb, k = 4))

## End(Not run)
```

`persona_audit` *Audit persona briefs for essentializing language and caricature*

Description

Read persona briefs back as text and flag the ways synthetic personas fail representationally. Two layers:

Usage

```
persona_audit(p_or_set, config = NULL, dimensions = NULL)

## S3 method for class 'persona_audit'
print(x, ...)
```

Arguments

`p_or_set`	A `persona_frame()`, a `persona_variants()` result (`persona_set`), or a list of `persona_frame` objects.
`config`	Optional generative `LLMR::llm_config()` for model scoring. When `NULL` (default), only the lexical layer runs and model scores are `NA`.
`dimensions`	Optional character vector naming the qualities to score (model layer); defaults to caricature, out-group homogeneity, and essentialism. Recorded in the judge prompt.
`x`	A `persona_audit`.
`...`	Ignored.

Details

- **Lexical** (always, no model): each brief is scanned against a small built-in lexicon of essentializing and demographic-as-destiny patterns. A brief that says a demographic *naturally* or *always* thinks something is flagged.
- **Model** (optional, when `config` is a generative `LLMR::llm_config()`): each brief is scored on caricature and essentialism on a 0–1 scale via `LLMR::llm_judge()`. Without a config these scores are `NA`.

The lexical layer is a screening pass, not a proof: a clean scan does not certify a brief is unbiased, and a hit may be a false positive in quoted speech. Treat the audit as evidence to read, alongside the briefs themselves.

Value

A tibble of class `persona_audit`, one row per persona, with columns `id`, `flag_lexical` (any lexical hit), `n_lexical_hits`, `caricature_score` (0–1 or `NA`), `essentialism_score` (0–1 or `NA`), and `notes`.

See Also

`persona_frame()`, `persona_variants()`, `diagnostics()`

Examples

```
set <- persona_variants(
  persona_frame("A small-business owner.", source = "synthetic"),
  vary = list(age = c("35", "60")))
persona_audit(set)
diagnostics(persona_audit(set))
## Not run:
cfg <- LLMR::llm_config("openai", "gpt-4o-mini")
persona_audit(set, config = cfg)   # adds model caricature scores

## End(Not run)
```

---

persona_frame *A persona as an auditable research object*

Description

A `persona_frame` bundles the brief a model reads (`text`, the *only* thing the model sees) with the provenance that makes a synthetic persona inspectable: its `source`, the `scope` conditions it is meant to hold under, the `attributes` that were varied to produce it, an optional `variant_of` parent hash, and a stable content `hash` (via `hash_persona()`). A frame is a drop-in replacement for a plain-string persona anywhere `agent()` accepts one: `as.character()` returns its `text`, and the Agent reads `text` directly while keeping the frame for provenance.

Usage

```
persona_frame(
  text,
  source = NULL,
  scope = NULL,
  attributes = NULL,
  variant_of = NULL,
  id = NULL
)

## S3 method for class 'persona_frame'
print(x, ...)
```

```
## S3 method for class 'persona_frame'
as.character(x, ...)
```

Arguments

text	The persona brief (character scalar): who this person is, what they want, how they speak. The only field the model sees.
source	Where the brief came from, e.g. "synthetic", "interview-grounded", "literature", or NULL if unrecorded.
scope	Optional named list of scope conditions the persona is meant to hold under (e.g. list(country = "US", year = 2024)); recorded as provenance, not enforced.
attributes	Optional named list of the dimensions varied to produce this brief (e.g. list(age = "52", risk = "cautious")).
variant_of	Optional parent persona hash this frame was derived from.
id	Optional explicit identifier; ignored for hashing (the hash is always content-derived) but kept on the object when supplied.
x	A persona_frame.
...	Ignored.

Details

Provenance, not authority: a persona is a prompt with a provenance record, never a claim that the model speaks for the people it sketches. Pair with persona_audit() before trusting a brief, and with persona_variants() to turn one frame into a designed set.

Value

An object of class persona_frame: a list with text, source, scope, attributes, variant_of, id, and hash.

See Also

persona_variants(), persona_audit(), hash_persona(), agent()

Examples

```
p <- persona_frame(
  "A retired schoolteacher who reads the local paper and distrusts polls.",
  source = "synthetic",
  scope = list(country = "US"))
print(p)
as.character(p)            # the brief, usable wherever a string persona is
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")
a <- agent("Voter", cfg, persona = p)   # the frame is accepted directly
a$persona_frame()$hash                  # provenance is preserved

## End(Not run)
```

`persona_variants` *Vary a persona along named dimensions*

Description

Turn one `persona_frame()` into a designed *set* of personas. Two modes:

Usage

```
persona_variants(p, vary, n = NULL, config = NULL)

## S3 method for class 'persona_set'
print(x, ...)
```

Arguments

`p`	A `persona_frame()` (the base).
`vary`	A named list of the dimensions to vary. Enumerated mode reads the level vectors (e.g. `list(age = c("28", "52"), risk = c("cautious", "tolerant"))`); generative mode reads only the names.
`n`	Number of briefs to generate (generative mode only). Ignored when enumerating.
`config`	Optional generative `LLMR::llm_config()`. When `NULL` (default), the set is enumerated offline.
`x`	A `persona_set`.
`...`	Ignored.

Details

- **Enumerated** (default, `config = NULL`): `vary` is a named list of level vectors and the result is their full Cartesian product. Each combination is rendered by appending the varied attributes to the base brief in plain, factual language (no stereotyping copula), so the design is legible and the base text is never rewritten.
- **Generative** (`config` is a generative `LLMR::llm_config()` and `n` is given): one structured call asks the model for `n` individuated briefs that vary `names(vary)`, under a hard-coded anti-essentialism instruction. Use this when you want fluent, non-templated briefs; audit the result with `persona_audit()`.

Varying a demographic attribute does not license writing a stereotype. The enumerated renderer states attributes flatly; the generative prompt forbids caricature. Neither mode certifies the output: it produces briefs to be inspected, not a population to be trusted.

Value

An object of class `persona_set`: a tibble with a `persona` list column of `persona_frame()` objects, an `id` column (each frame's hash), a `variant_of` column (the base hash), and one column per varied attribute.

See Also

persona_frame(), persona_audit()

Examples

```
base <- persona_frame("A first-time voter in a swing district.",
                      source = "synthetic")
set <- persona_variants(base, vary = list(age = c("19", "24"),
                                          leaning = c("undecided", "left")))
set
set$persona[[1]]$attributes
## Not run:
cfg <- LLMR::llm_config("openai", "gpt-4o-mini")
gen <- persona_variants(base, vary = list(age = NA, occupation = NA),
                        n = 5, config = cfg)
persona_audit(gen)

## End(Not run)
```

---

replay_run *Replay a workflow run, verifying state hashes*

Description

Re-executes the graph from the original input and compares each node's resulting state hash to the recorded one. `verify = "structural"` (default) checks deterministic nodes (functions/evaluators) exactly and, for model (agent) nodes, does not require the sampled text to match (model output is nondeterministic); `verify = "strict"` requires every node to match (sound only for fully deterministic graphs or archive-served calls). A mismatch raises `llmragent_replay_mismatch` naming the first divergent node.

Usage

```
replay_run(
  x,
  wf,
  verify = c("structural", "strict"),
  max_steps = 64L,
  quiet = TRUE,
  ...
)
```

Arguments

x	A checkpoint_dir or agent_workflow_run.
wf	The agent_workflow.
verify	"structural" or "strict".
max_steps, quiet, ...	As in run_workflow().

Details

A run containing a human_gate() replays up to the gate and pauses there, verifying the executed nodes before it; a pause is not an executed node, so its replay_match is NA and it occupies no comparison position.

Value

An agent_workflow_run for the replay (its steps carries a replay_match column).

See Also

run_workflow()

---

report.agent_experiment

Draft a short report for an agent experiment

Description

A compact account of an agent_experiment() result: a header with the cell and failure counts, then a per-condition breakdown over the design columns (every column that is not result, error, duration, or rep), reporting each condition's cell count and failures.

Usage

```
## S3 method for class 'agent_experiment'
report(x, ...)
```

Arguments

x	An agent_experiment() result (a tibble of class agent_experiment).
...	Unused.

Value

An object of class agent_report (a character vector with a print method).

See Also

diagnostics(), agent_experiment()

`report.agent_run` *Draft a methods-section report for an agent run*

Description

Composes a short, citable account of a run: a design header (kind, run id, the participating agents, and a compact workflow summary), the model and token paragraph drafted by `LLMR::report()` over the run's call-level rows, and a one-line limits note stating that the results are model-conditioned and are not estimates of a human population.

Usage

```
## S3 method for class 'agent_run'
report(x, ...)

## S3 method for class 'Agent'
report(x, ...)
```

Arguments

`x`	An object accepted by `as_agent_run()`.
`...`	May include `task`, a one-clause description of what the model was asked to do (spliced into the methods paragraph).

Value

An object of class `agent_report`: a character vector with a print method that `cat()`s the lines.

See Also

`diagnostics()`, `LLMR::report()`

Examples

```
## Not run:
a <- agent("Aria", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
a$chat("Hello")
report(a, task = "to answer a factual question")

## End(Not run)
```

reset.Agent *Clear an agent's memory*

Description

`LLMR::reset(agent)` delegates to the agent's own `agent$reset()` method, so the two agree: both clear the agent's conversation memory, returning the agent invisibly. Use it to reuse one configured agent across independent trials without leaking state between them.

Usage

```
## S3 method for class 'Agent'
reset(x, ...)
```

Arguments

`x`	An Agent.
`...`	Unused.

Value

The agent, invisibly.

See Also

Agent

Examples

```
## Not run:
a <- agent("Aria", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
a$chat("Remember the number 7.")
LLMR::reset(a)        # same as a$reset()

## End(Not run)
```

resume_run *Resume a paused run after a tool-approval decision*

Description

Rebuilds the paused agent from the checkpoint, applies the recorded decision (approve / reject / edit), and continues the tool loop to completion. The resumed work keeps the original accounting, so the run reads as one continuous exchange.

Usage

```
resume_run(checkpoint, ...)

## S3 method for class 'agent_resume_result'
as.character(x, ...)

## S3 method for class 'agent_resume_result'
print(x, ...)
```

Arguments

checkpoint	A llmragent_checkpoint with a decision recorded by approve_tool_call().
...	Reserved.
x	An agent_resume_result object.

Value

An object of class agent_resume_result with text (the final reply), agent (the rebuilt agent, or NULL), and checkpoint (the checkpoint that was resumed).

See Also

human_gate(), approve_tool_call()

---

resume_workflow *Resume a paused or failed workflow run*

Description

Continues from the last good checkpoint without rerunning completed nodes. For a run paused at a human gate, supply approve = TRUE to proceed (or FALSE to stop). For a failed run, resume retries from the failed node.

Usage

```
resume_workflow(
  x,
  wf = NULL,
  approve = TRUE,
  max_steps = 64L,
  quiet = TRUE,
  ...
)
```

Arguments

x	A paused/failed agent_workflow_run, its checkpoint, or a checkpoint_dir path (for a failed run, also pass wf).
wf	The agent_workflow (required when x is a bare checkpoint_dir from a failed run, which carries no embedded workflow).
approve	For a gated pause: TRUE to continue past the gate.
max_steps	Loop guard for the continuation.
quiet	If FALSE, print progress.
...	Passed to agent nodes.

Value

An agent_workflow_run.

See Also

run_workflow(), fork_workflow(), replay_run()

---

robustness-axes *Robustness perturbation axes*

Description

Helpers that declare a perturbation axis for agent_robustness(). Each returns a small spec the battery expands into design cells and applies to the inputs through the perturb argument of your run_fn. Bare vectors work too (vary = list(model = c("a","b"))).

Usage

```
vary_models(...)

vary_temperature(...)

vary_prompt(..., prompt = NULL, paraphrase = NULL, config = NULL)

vary_persona(...)

vary_option_order(..., seed = 110)
```

Arguments

...	For vary_models/vary_temperature, the levels (model names or temperatures). For vary_prompt, either named template strings or paraphrase = n with config = to generate n paraphrases. For vary_persona, persona variants (strings or a persona_set). For vary_option_order, the orders ("as_is", "reverse", "random").

prompt	For vary_prompt(paraphrase=): the actual prompt text to paraphrase. The paraphrase set is generated once at build time and is the axis's levels (including the original as the baseline), so run_fn's perturb$prompt(x) returns the cell's paraphrase, not a placeholder.
paraphrase	For vary_prompt: number of paraphrases to generate.
config	For vary_prompt(paraphrase=): a generative config used once to draft the paraphrases (hashed into the manifest).
seed	For vary_option_order: RNG seed for the random permutation.

Value

An agent_axis object.

See Also

agent_robustness()

---

run_workflow *Run a workflow*

Description

Executes the graph from its entry node, threading one explicit state list, writing a checkpoint (state snapshot + event line) after each node. Stops at a terminal node (no outgoing edge taken), when a human gate pauses the run, or when max_steps node executions is reached.

Usage

```
run_workflow(
  wf,
  input = NULL,
  state = NULL,
  checkpoint_dir = NULL,
  max_steps = 64L,
  quiet = TRUE,
  ...
)
```

Arguments

wf	An agent_workflow.
input	The initial input, placed at state$input.
state	Optional initial state list (merged with input).
checkpoint_dir	Optional directory for checkpoints (state RDS + a run.jsonl event log + a cursor.json). When NULL, a temporary directory is used so resume/replay still work within the session.

max_steps	Maximum node executions before stopping (loop guard).
quiet	If FALSE, print one line per node.
...	Passed to agent nodes' reply().

Value

An object of class agent_workflow_run: a list with status ("done", "paused", or "failed"), state, checkpoint_dir, steps (a tibble of node/status/state_hash), and (when paused) checkpoint.

See Also

resume_workflow(), fork_workflow(), replay_run()

---

save_agent *Save an agent to disk*

Description

Writes the agent's name, persona, config, memory contents, guardrails, and trace to an RDS file. Tools are functions and are not serialized; re-attach them at load time via load_agent(tools = ...). Guardrails are serialized (their check functions travel through RDS) and restored automatically.

Usage

```
save_agent(x, path)
```

Arguments

x	An Agent.
path	File path (.rds).

Details

A config carrying a literal API key (one passed as a string rather than the usual environment-variable reference) is refused: saving it would write the key to disk. Rebuild the config from an environment variable.

Value

path, invisibly.

See Also

load_agent()

Examples

```
## Not run:
cfg <- LLMR::llm_config("groq", "openai/gpt-oss-20b")
a <- agent("Ada", cfg, persona = "Terse.")
a$chat("Remember this: the project deadline is March 3.")
save_agent(a, "ada.rds")

# a later session, any machine with GROQ_API_KEY set:
ada <- load_agent("ada.rds")
ada$chat("When is the deadline?")   # the memory came along

## End(Not run)
```

`view_run` *View a run as a self-contained HTML inspector*

Description

`view_run()` renders an agent_run (or anything `as_agent_run()` accepts) into a single self-contained HTML file: a header (kind, run id, short manifest hash, agents, creation time, total calls and tokens) followed by one table per grain (utterance, event, call, tool). Event rows carry an HTML anchor keyed by their `span_id`, and transcript and call cells that name a span link to it, so a reader can trace an utterance to the model call, span event, and tool output that produced it.

Usage

```
view_run(run, output = NULL, open = interactive())
```

Arguments

`run`	An Agent or any object accepted by `as_agent_run()`.
`output`	Path to write. Defaults to a temp file with extension `.html`. A missing parent directory is created.
`open`	If `TRUE` (the default in an interactive session) the file is opened with `utils::browseURL()` after writing.

Details

The inspector is a read-only view: it visualizes the substrate already captured by the run and derives nothing. It prefers `htmltools` when that package is installed; otherwise it builds the same content by hand with no optional dependencies. If no HTML page can be produced at all, it writes a structured text summary so the call still yields a file.

Value

The output path, invisibly, with class `agent_inspector_path` (its print method reports where the file was written).

See Also

as_agent_run(), agent_manifest(), report()

Examples

```
## Not run:
a <- agent("Ada", LLMR::llm_config("groq", "openai/gpt-oss-20b"))
a$chat("Tell me something true.")
view_run(a)

## End(Not run)
```

`workflow_from_pipeline` *Express an agent pipeline as a workflow*

Description

Builds the linear graph equivalent of agent_pipeline() (node i is agent i, each feeding the next). The original agent_pipeline() keeps its own simple implementation; this exists to show the graph can express it and to let a linear pipeline gain checkpointing when wanted.

Usage

```
workflow_from_pipeline(agents)
```

Arguments

agents	A list of Agents, in order.

Value

An agent_workflow.

See Also

agent_pipeline(), run_workflow()

Index