

Package: HydraR (via r-universe)

September 17, 2026

Type Package

Title Stateful Agentic Orchestration for Scientific Reproducibility

Version 0.2.4

Description A high-performance framework for orchestrating complex agentic workflows in R, specifically designed for scientific reproducibility and auditability. HydraR provides a robust, state-managed engine for Directed Acyclic Graphs (DAGs) and iterative state machines, prioritizing CLI-native LLM interactions (e.g., Gemini, Claude, Copilot) and hardened state persistence (DuckDB/SQLite). It features isolated execution via Git Worktrees for parallel file-modifying tasks, autonomous quality control through integrated auditors, and human-readable visualization using Mermaid.js. Ideal for creating auditable, resumable research assistants and complex bioinformatics pipelines.

URL <https://github.com/APAF-bioinformatics/HydraR>,
<https://APAF-bioinformatics.github.io/HydraR/>

BugReports <https://github.com/APAF-bioinformatics/HydraR/issues>

License LGPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports uuid, DBI, digest, furr, igraph, jsonlite, purrr, R6, yaml,
httr2, base64enc

Suggests testthat (>= 3.0.0), duckdb, knitr, rmarkdown, future,
devtools, withr, reticulate, DiagrammeR, ggplot2,
DiagrammeRsvg, lintr, rsvg, mockery, RSQLite, htmltools

VignetteBuilder knitr

Config/testthat/edition 3

Collate 'checkpointer.R' 'dag.R' 'driver.R' 'driver_registry.R'
 'drivers_api.R' 'drivers_cli.R' 'factory.R' 'git_cleanup.R'
 'init_duckdb.R' 'logic_node.R' 'map_node.R'
 'merge_harmonizer.R' 'mermaid_parser.R' 'message_log.R'
 'node.R' 'node_languages.R' 'node_llm.R' 'observer_node.R'
 'registry.R' 'router_node.R' 'state.R' 'state_restricted.R'
 'tools.R' 'utils.R' 'validation.R' 'worktree.R' 'zzz.R'

Config/pak/sysreqs libglpk-dev libxml2-dev libssl-dev

Repository <https://ropensci-staging.r-universe.dev>

Date/Publication 2026-04-23 14:30:55 UTC

RemoteUrl <https://github.com/APAF-bioinformatics/HydraR>

RemoteRef HEAD

RemoteSha ceb3c632cd464cf6ea6c5c6b078d713b47af5ca0

Contents

add_llm_node	3
add_logic_node	5
AgentBashNode	6
AgentDAG	7
AgentDriver	15
AgentLLMNode	18
AgentLogicNode	20
AgentMapNode	21
AgentNode	23
AgentObserverNode	25
AgentPythonNode	26
AgentRouterNode	28
AgentState	30
AgentTool	32
AnthropicAPIDriver	34
AnthropicCLIDriver	36
auto_node_factory	38
Checkpoint	39
cleanup_jules_branches	40
ConflictResolver	41
CopilotCLIDriver	43
create_merge_harmonizer	44
dag_add_llm_node	45
dag_add_logic_node	46
dag_create	47
DriverRegistry	48
DuckDBMessageLog	50
DuckDBSaver	52
extract_r_code_advanced	53
format_toolset	54

GeminiAPIDriver	55
GeminiCLIDriver	57
GeminiImageDriver	59
get_agent_roles	61
get_default_driver	62
get_driver_registry	62
get_logic	63
get_role	64
get_role_prompt	64
init_bot_history	65
is_named_list	66
JSONLMessageLog	66
list_logic	68
load_workflow	69
MemoryMessageLog	70
MemorySaver	71
mermaid_to_dag	72
MessageLog	73
OllamaDriver	74
OpenAIAPIDriver	77
OpenAICodexCLIDriver	79
parse_mermaid	80
RDSSaver	81
reducer_append	82
reducer_merge_list	83
register_logic	84
register_role	85
render_workflow_file	86
resolve_default_driver	87
RestrictedState	88
set_default_driver	90
spawn_dag	91
standard_node_factory	92
validate_workflow_file	92
validate_workflow_full	93
WorktreeManager	94
Index	98

add_llm_node

Create an LLM Agent Node

Description

A convenience wrapper to instantiate an AgentLLMNode. Useful for functional-style DAG construction.

Usage

```
add_llm_node(id, role, driver, model = NULL, cli_opts = list(), ...)
```

Arguments

<code>id</code>	String. A unique identifier for the node within the DAG.
<code>role</code>	String. The system prompt or identity the LLM should assume (e.g., "Python Developer").
<code>driver</code>	AgentDriver. An R6 driver object (e.g., GeminiCLIDriver) that handles the LLM API.
<code>model</code>	String. Optional model name override (e.g., "gpt-4"). Defaults to driver default.
<code>cli_opts</code>	List. Optional parameters passed to the underlying CLI or API call.
<code>...</code>	Additional arguments. Passed directly to the AgentLLMNode\$new() constructor. Useful for setting tools, prompt_builder, or output_path.

Value

An AgentLLMNode object.

Examples

```
## Not run:
# Standard API-based agent with model overrides and safety parameters
# Use Sys.getenv to securely load your API key from .Renvirom
driver <- AnthropicAPIDriver$new(api_key = Sys.getenv("ANTHROPIC_API_KEY"))
node_coder <- add_llm_node(
  id = "optimizer",
  role = "You are an expert at optimizing R code for performance.",
  driver = driver,
  model = "claude-3-opus-20240229",
  cli_opts = list(temperature = 0, max_tokens = 1024),
  output_path = "output/optimized.R"
)

# CLI-based agent with sandbox permissions and tool discovery
gemini <- GeminiCLIDriver$new()
node_researcher <- add_llm_node(
  id = "researcher",
  role = "Find bioinformatics papers on 'single-cell RNA-seq' using PubMed.",
  driver = gemini,
  cli_opts = list(allowed_tools = "pubmed_search,read_pdf", yolo = TRUE)
)

## End(Not run)
```

add_logic_node	<i>Create an R Logic Node</i>
----------------	-------------------------------

Description

A convenience wrapper to instantiate an AgentLogicNode. Nodes created this way execute pure R code rather than calling an LLM.

Usage

```
add_logic_node(id, logic_fn, ...)
```

Arguments

id	String. A unique identifier for the node.
logic_fn	Function. An R function that accepts an AgentState object as its first argument and returns a list with at least status and output.
...	Additional arguments. Passed directly to the AgentLogicNode\$new() constructor.

Value

An AgentLogicNode object.

Examples

```
## Not run:
# 1. Validation Logic: Check if previous output meets quality thresholds
quality_gate <- function(state) {
  results <- state$get("researcher")
  if (length(results$papers) >= 5) {
    list(status = "success", output = list(proceed = TRUE))
  } else {
    # 'pause' status can trigger a human-in-the-loop or a retry loop
    list(status = "pause", output = list(reason = "insufficient results"))
  }
}
node_gate <- add_logic_node("quality_gate", quality_gate)

# 2. Data Transformation Logic: Clean and format LLM output
cleaner <- function(state) {
  raw_text <- state$get("coder")
  clean_code <- extract_r_code_advanced(raw_text)
  list(status = "success", output = list(code = clean_code))
}
node_clean <- add_logic_node("cleaner", cleaner)

## End(Not run)
```

AgentBashNode	<i>Bash Execution Node</i>
---------------	----------------------------

Description

Executes a raw bash script. Can run within an isolated worktree if configured.

Value

An 'AgentBashNode' object.

Super class

[HydraR::AgentNode](#) -> AgentBashNode

Public fields

`script` Character string or Function. The bash script to execute.

`env_vars` Named list. Environment variables to inject.

Methods**Public methods:**

- [AgentBashNode\\$new\(\)](#)
- [AgentBashNode\\$run\(\)](#)
- [AgentBashNode\\$clone\(\)](#)

Method new(): Initialize Bash Node

Usage:

```
AgentBashNode$new(id, label = NULL, script, env_vars = list(), params = list())
```

Arguments:

`id` Node ID

`label` Node Label

`script` String or Function(state) returning a string.

`env_vars` Named list of environment variables

`params` List of standard Node parameters Run Bash Execution

Method run():

Usage:

```
AgentBashNode$run(state, working_dir = NULL)
```

Arguments:

`state` AgentState object.

`working_dir` Directory context for execution.

Returns: List with output, status_code, and success flag.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Simple static bash command
node_simple <- AgentBashNode$new(id = "bash_ls", script = "ls -la")

# 2. Dynamic bash script using state values and environment variables
# The script is generated at runtime based on the current AgentState.
dynamic_script <- function(state) {
  input_file <- state$get("raw_data_path")
  sprintf("cat %s | grep 'ERR' > failure_log.txt", input_file)
}

node_dynamic <- AgentBashNode$new(
  id = "bash_filter",
  script = dynamic_script,
  env_vars = list(THRESHOLD = "100", CPU_LIMIT = "4")
)

# Execute the node
res <- node_dynamic$run(state = AgentState$new(list(raw_data_path = "data.csv")))
message(res$output)

## End(Not run)
```

AgentDAG

Agent Graph R6 Class

Description

The core orchestrator in HydraR, AgentDAG defines and executes a Directed Graph of AgentNode objects. It supports:

- **Pure DAGs:** Parallel execution using `furrr`.
- **Cycles & Loops:** Iterative execution via conditional edges.
- **State Isolation:** Parallel branch execution in isolated git worktrees.
- **Persistence:** Automatic state checkpointing and restoration.

Value

An AgentDAG R6 object.

Public fields

nodes List. Named list of AgentNode objects.
edges List. Pending edge definitions to be bound.
conditional_edges List. Conditional transition logic.
error_edges List. Failover transition logic.
results List. Execution results for each node.
trace_log List. Execution telemetry and tracing.
graph igraph. Internal graph representation.
start_node String. Explicit entry point for cycles.
state AgentState. Centralized state object.
message_log MessageLog. Optional audit log.
worktree_manager WorktreeManager. Optional isolation manager.

Methods**Public methods:**

- `AgentDAG$new()`
- `AgentDAG$set_start_node()`
- `AgentDAG$add_node()`
- `AgentDAG$add_edge()`
- `AgentDAG$add_conditional_edge()`
- `AgentDAG$add_error_edge()`
- `AgentDAG$run()`
- `AgentDAG$.run_linear()`
- `AgentDAG$.run_iterative()`
- `AgentDAG$plot()`
- `AgentDAG$get_terminal_nodes()`
- `AgentDAG$get_start_nodes()`
- `AgentDAG$compile()`
- `AgentDAG$save_trace()`
- `AgentDAG$from_mermaid()`
- `AgentDAG$clone()`

Method `new()`: Initialize AgentDAG

Usage:

`AgentDAG$new()`

Returns: A new AgentDAG instance with empty node and edge lists.

Method `set_start_node()`: Set Start Node(s) Explicitly defines the entry point(s) of the graph. If not called, the engine defaults to nodes with an in-degree of 0. Required for graphs with cycles or where execution must start at a specific node.

Usage:

```
AgentDAG$set_start_node(node_ids)
```

Arguments:

`node_ids` Character vector. A vector of one or more node IDs. These nodes must already exist in the DAG (added via `add_node`).

Returns: The AgentDAG object (invisibly).

Method `add_node()`: Add a Node Registers an AgentNode object into the graph.

Usage:

```
AgentDAG$add_node(node)
```

Arguments:

`node` AgentNode. An instance of AgentNode or a subclass (e.g., AgentLLMNode). The ID of the node must be unique within the DAG.

Returns: The AgentDAG object (invisibly).

Method `add_edge()`: Add an Edge Creates a directed connection between nodes. Also supports specialized edge types (Error, Test, Fail) via labeling.

Usage:

```
AgentDAG$add_edge(from, to, label = NULL)
```

Arguments:

`from` String or Character vector. The source node ID(s).

`to` String. The destination node ID.

`label` String. Optional label for the edge. In HydraR, certain labels trigger logic:

- "error" or "failover": Creates an error edge.
- "Test": Creates a conditional edge (true path).
- "Fail": Creates a conditional edge (false path).

Returns: The AgentDAG object (invisibly).

Method `add_conditional_edge()`: Add a Conditional Edge (Loop Support) Adds branching logic to the graph. After the from node executes, the test function is evaluated against the node's result.

Usage:

```
AgentDAG$add_conditional_edge(
  from,
  test = NULL,
  if_true = NULL,
  if_false = NULL
)
```

Arguments:

`from` String. The ID of the source node.

`test` Function. A predicate function that takes the node result list and returns TRUE or FALSE.

`if_true` String. Optional ID of the node to execute if the test passes.

`if_false` String. Optional ID of the node to execute if the test fails.

Returns: The AgentDAG object (invisibly).

Method `add_error_edge()`: Add an Error Edge (Failover Support)

Usage:

```
AgentDAG$add_error_edge(from, to)
```

Arguments:

`from` String node ID.

`to` String node ID.

Returns: The AgentDAG object (invisibly).

Method `run()`: Run the Graph The primary execution engine for the AgentDAG. It manages the orchestration lifecycle, including state initialization / recovery, worktree setup for isolation, and routing between nodes. It automatically switches between linear (parallel-capable) and iterative execution modes based on the graph's complexity.

Usage:

```
AgentDAG$run(
  initial_state = NULL,
  max_steps = 25,
  checkpointer = NULL,
  thread_id = NULL,
  resume_from = NULL,
  use_worktrees = FALSE,
  repo_root = getwd(),
  cleanup_policy = "auto",
  fail_if_dirty = TRUE,
  packages = c("withr"),
  ...
)
```

Arguments:

`initial_state` List, AgentState, or String. The starting data for the workflow. Can be a named list of R objects, an existing AgentState instance, or a path to a checkpoint (if supported). Required unless resuming from a checkpointer.

`max_steps` Integer. The maximum number of node executions allowed in a single `run()` call. Prevents infinite loops in cyclic graphs. Default is 25.

`checkpointer` Checkpointer. An optional Checkpointer R6 object. If provided, the state is automatically saved after every node execution.

`thread_id` String. A unique identifier for this execution "thread" or session. Required if using a checkpointer to resolve the correct state from storage.

`resume_from` String. Optional node ID. If provided (or found in a checkpoint), execution will skip completed nodes and start from this point.

`use_worktrees` Logical. Enable branch isolation. If TRUE, parallel branches are executed in separate git worktrees, preventing file-system conflicts between agents.

`repo_root` String. Path to the master git repository. Required if `use_worktrees` is TRUE.

`cleanup_policy` String. One of "auto" (default), "none", or "aggressive". Determines how worktrees are removed after execution.

`fail_if_dirty` Logical. If TRUE, execution fails if the `repo_root` has uncommitted changes (recommended for reproducibility when using worktrees).

`packages` Character vector. A list of R packages to load on parallel worker nodes (passed to `furrr`).

`...` Additional arguments. Passed down to individual `node$run()` calls.

Returns: A list containing:

- `results`: A named list of each node's output.
- `state`: The final `AgentState` object.
- `status`: "completed" or "paused".

Internal: Linear DAG Execution

Method `.run_linear()`:

Usage:

```
AgentDAG$.run_linear(
  max_steps = 25,
  checkpointer = NULL,
  thread_id = NULL,
  resume_from = NULL,
  node_ids = NULL,
  step_count = 0,
  fail_if_dirty = TRUE
)
```

Arguments:

`max_steps` Integer.

`checkpointer` Checkpointer object.

`thread_id` String thread ID.

`resume_from` Node ID(s) to resume from.

`node_ids` Character vector of nodes to run.

`step_count` Integer current total step count.

`fail_if_dirty` Logical.

Returns: Execution result list. Internal: Iterative Execution

Method `.run_iterative()`:

Usage:

```
AgentDAG$.run_iterative(
  max_steps,
  checkpointer = NULL,
  thread_id = NULL,
  resume_from = NULL,
  step_count = 0,
  fail_if_dirty = TRUE,
  packages = c("withr")
)
```

Arguments:

`max_steps` Integer.

`checkpointer` Checkpointer object.

thread_id String.
 resume_from String.
 step_count Integer.
 fail_if_dirty Logical.
 packages Character vector. Packages to load in parallel workers. Visualize the Graph

Method `plot()`: Generates a visual representation of the DAG using Mermaid or DOT syntax.

Usage:

```

AgentDAG$plot(
  type = "mermaid",
  status = FALSE,
  details = FALSE,
  include_params = NULL,
  show_edge_labels = TRUE
)

```

Arguments:

type String. Either "mermaid" (default) for web-native rendering or "grViz" for DiagrammeR/Graphviz.
 status Logical. If TRUE, colors nodes based on the results in the current trace log (Green: Success, Red: Failed, Yellow: Paused).
 details Logical. If TRUE, injects node parameters into the labels.
 include_params Character vector. Optional whitelist of parameter names to display when details is TRUE.
 show_edge_labels Logical. Whether to display labels (e.g., "Test", "Fail") on edges.

Returns: The Mermaid/DOT source string (invisibly).

Examples:

```

\dontrun{
dag <- dag_create()
dag$add_node(AgentNode$new("A"))
dag$add_node(AgentNode$new("B"))
dag$add_edge("A", "B", label = "proceed")

# Generate Mermaid string
m_src <- dag$plot(type = "mermaid", show_edge_labels = TRUE)
cat(m_src)
}

```

Method `get_terminal_nodes()`: Get Terminal Nodes Identifies nodes with no outgoing edges.

Usage:

```
AgentDAG$get_terminal_nodes()
```

Returns: Character vector of node IDs.

Method `get_start_nodes()`: Get Start Nodes (Roots) Identifies nodes with no incoming edges.

Usage:

`AgentDAG$get_start_nodes()`

Returns: Character vector of node IDs. Compile the Graph

Method `compile()`: Rebuilds the internal igraph representation and performs validation checks such as cycle detection, start node disambiguation, and reachability analysis. This must be called before `run()`.

Usage:

`AgentDAG$compile()`

Details: Compilation will throw errors if:

- Cycles are found in a pure DAG (no conditional edges).
- Multiple root nodes are found without an explicit `start_node`.
- Node IDs used in edges do not exist.

Returns: The AgentDAG object (invisibly). Save the Execution Trace

Method `save_trace()`: Exports the detailed telemetry from the last execution (durations, status, outputs) to a JSON file.

Usage:

`AgentDAG$save_trace(file = "dag_trace.json")`

Arguments:

`file` String. Path to the output JSON file. Defaults to `"dag_trace.json"`.

Returns: The AgentDAG object (invisibly). Create Graph from Mermaid

Method `from_mermaid()`: A static-like method to populate a DAG from a Mermaid string using a custom node factory.

Usage:

`AgentDAG$from_mermaid(mermaid_str, node_factory)`

Arguments:

`mermaid_str` String. A valid Mermaid graph definition (e.g., `"graph TD; A-->B"`).

`node_factory` Function. A closure mapping Mermaid labels and parameters to `AgentNode` instances. See [auto_node_factory](#) for the standard implementation.

Returns: The AgentDAG object (invisibly).

Examples:

```
\dontrun{
# Use a custom factory to map all nodes to Logic nodes
simple_factory <- function(id, label, params) {
  AgentLogicNode$new(id = id, logic_fn = function(s) list(status="ok"))
}

dag <- AgentDAG$new()
dag$from_mermaid("graph LR; Start-->End", simple_factory)
}
```

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

Usage:

`AgentDAG$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Define a complex DAG with branching, loops, and custom state
# Initializing the orchestrator
dag <- AgentDAG$new()

# Define logic for data retrieval and quality control
register_logic("fetcher", function(state) list(status = "success", output = "raw data"))
register_logic("gate", function(state) {
  if (nchar(state$fetcher) > 5) list(status = "true") else list(status = "false")
})

# 2. Programmatic Node Construction
dag$add_node(AgentLogicNode$new("start", logic_fn = get_logic("fetcher")))
dag$add_node(AgentLogicNode$new("check", logic_fn = get_logic("gate")))

# 3. Connecting nodes with status-based edges
# Errors in 'start' go to a terminal cleanup node
dag$add_edge("start", "check")
dag$add_conditional_edge("check",
  test = function(res) res$status == "true",
  if_true = "process", if_false = "retry"
)

# 4. Multi-agent execution context
# Compiling and running with initial state
dag$compile()
final_state <- dag$run(initial_state = list(job_id = "123"))

# Checking results
print(final_state$get_all())

## End(Not run)

## -----
## Method `AgentDAG$plot`
## -----

## Not run:
dag <- dag_create()
dag$add_node(AgentNode$new("A"))
dag$add_node(AgentNode$new("B"))
dag$add_edge("A", "B", label = "proceed")

# Generate Mermaid string
m_src <- dag$plot(type = "mermaid", show_edge_labels = TRUE)
cat(m_src)

## End(Not run)

## -----
## Method `AgentDAG$from_mermaid`
```

```
## -----

## Not run:
# Use a custom factory to map all nodes to Logic nodes
simple_factory <- function(id, label, params) {
  AgentLogicNode$new(id = id, logic_fn = function(s) list(status="ok"))
}

dag <- AgentDAG$new()
dag$from_mermaid("graph LR; Start-->End", simple_factory)

## End(Not run)
```

AgentDriver

Agent Driver R6 Class

Description

Abstract base class for CLI-based LLM drivers.

Value

An ‘AgentDriver’ R6 object.

Public fields

`id` String. Unique identifier for the driver.

`provider` String. Provider name (e.g., "google", "ollama").

`model_name` String. The specific model identifier.

`supported_opts` Character vector. Allowed CLI option names.

`validation_mode` String. Either "warning" or "strict".

`working_dir` String. Optional path to the working directory/worktree.

`repo_root` String. Path to the main repository root (for root-locked CLIs).

Methods

Public methods:

- `AgentDriver$new()`
- `AgentDriver$get_capabilities()`
- `AgentDriver$exec_in_dir()`
- `AgentDriver$filter_llm_noise()`
- `AgentDriver$call()`
- `AgentDriver$validate_no_injection()`
- `AgentDriver$validate_cli_opts()`
- `AgentDriver$format_cli_opts()`

- `AgentDriver$clone()`

Method new(): Initialize AgentDriver

Usage:

```
AgentDriver$new(
  id,
  provider = "unknown",
  model_name = "unknown",
  validation_mode = "warning",
  working_dir = NULL,
  repo_root = NULL
)
```

Arguments:

`id` Unique identifier.

`provider` String. Provider name.

`model_name` String. Model identifier.

`validation_mode` String. "warning" or "strict".

`working_dir` String. Optional working directory.

`repo_root` String. Path to the main repository root. Get Driver Capabilities

Method get_capabilities():

Usage:

```
AgentDriver$get_capabilities()
```

Returns: List of logicals. Execute Command in Working Directory

Method exec_in_dir(): Safely executes a system command within the specified 'working_dir' using 'withr::with_dir' to ensure the original CWD is restored.

Usage:

```
AgentDriver$exec_in_dir(command, args, ...)
```

Arguments:

`command` String. The command to run.

`args` Character vector. Command arguments.

`...` Additional arguments passed to system2.

Returns: Result of system2 call. Filter CLI Noise from LLM Output

Method filter_llm_noise(): Removes common CLI-injected headers, keychain warnings, or MCP status messages that can corrupt the generated model content.

Usage:

```
AgentDriver$filter_llm_noise(text)
```

Arguments:

`text` String or Character Vector. Raw output from the CLI.

Returns: Character vector of cleaned lines. Call the LLM

Method call():

Usage:

```
AgentDriver$call(  
  prompt,  
  model = NULL,  
  system_prompt = NULL,  
  cli_opts = list(),  
  ...  
)
```

Arguments:

`prompt` String. The prompt to send.
`model` String. Optional model override.
`system_prompt` String. Optional system prompt.
`cli_opts` List. Named list of CLI options.
... Additional arguments.

Returns: String. Cleaned response from the LLM. Validate No Command Injection

Method `validate_no_injection()`: Portably prevents command injection by rejecting inputs with shell metacharacters.

Usage:

```
AgentDriver$validate_no_injection(x)
```

Arguments:

`x` String or character vector.

Returns: The original string invisibly, or throws an error. Validate CLI Options

Method `validate_cli_opts()`:

Usage:

```
AgentDriver$validate_cli_opts(cli_opts)
```

Arguments:

`cli_opts` List. Named list to validate.

Returns: Invisible TRUE if valid. Format CLI Options

Method `format_cli_opts()`:

Usage:

```
AgentDriver$format_cli_opts(cli_opts = list())
```

Arguments:

`cli_opts` List. Named list to format.

Returns: Character vector of CLI flags.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
driver <- AgentDriver$new(id = "test", provider = "mock")

## End(Not run)
```

AgentLLMNode

*Agent LLM Node R6 Class***Description**

A specialized AgentNode that leverages a Large Language Model (LLM) to generate outputs from prompts. It manages prompt construction by combining a persistent role (system prompt) with dynamic context from the AgentState. It also handles tool injection and automatic context discovery from local files (e.g., agents.md, skills.md).

Value

An AgentLLMNode object.

Super class

[HydraR::AgentNode](#) -> AgentLLMNode

Public fields

role String. System prompt/role for the agent.
 model String. Default model.
 driver AgentDriver object.
 output_format String. Output expectation.
 cli_opts List. Default CLI options for the driver.
 prompt_builder Function(state) -> String.
 tools List of AgentTool objects.
 agents_files Character vector. Paths to agents context files.
 skills_files Character vector. Paths to skills context files.

Methods**Public methods:**

- [AgentLLMNode\\$new\(\)](#)
- [AgentLLMNode\\$run\(\)](#)
- [AgentLLMNode\\$swap_driver\(\)](#)
- [AgentLLMNode\\$clone\(\)](#)

Method new(): Initialize AgentLLMNode

Usage:

```
AgentLLMNode$new(
  id,
  role,
  driver,
  model = NULL,
  cli_opts = list(),
  prompt_builder = NULL,
  tools = list(),
  label = NULL,
  params = list(),
  agents_files = NULL,
  skills_files = NULL
)
```

Arguments:

`id` String. Unique identifier for the node.

`role` String. The primary system prompt or persona the LLM should assume.

`driver` AgentDriver. An instance of an AgentDriver subclass (CLI or API based).

`model` String. Optional. The specific model to use (overrides driver default).

`cli_opts` List. Optional. Named list of parameters for the LLM call (e.g., temperature).

`prompt_builder` Function. Optional. A function that takes an AgentState and returns a string prompt. If omitted, the node serializes the entire state as JSON.

`tools` List. A list of AgentTool objects available for the agent to use.

`label` String. Human-readable name for visualization.

`params` List. Additional configuration (e.g., `output_format="r"`).

`agents_files` Character vector. Optional paths to markdown files containing agent interaction guidelines.

`skills_files` Character vector. Optional paths to markdown files containing specialized tool instructions. Run the LLM Node

Method run(): Executes the LLM call. This method handles prompt construction, tool injection, context file discovery, and driver invocation.

Usage:

```
AgentLLMNode$run(state, ...)
```

Arguments:

`state` AgentState. The centralized state object for the workflow.

`...` Additional arguments. Passed through to the driver's `call()` method.

Returns: A list containing status, output (the LLM response), raw (the full driver response), and meta-information. Swap Driver at Runtime

Method swap_driver():

Usage:

```
AgentLLMNode$swap_driver(driver)
```

Arguments:

driver AgentDriver object or String ID.

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

Usage:

AgentLLMNode\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# NOTE: Set ANTHROPIC_API_KEY in your .Renviron file
driver <- AnthropicAPIDriver$new()

# Create a node with a prompt builder that pulls from state
node <- AgentLLMNode$new(
  id = "summarizer",
  role = "You are a concise summarizer.",
  driver = driver,
  prompt_builder = function(state) {
    sprintf("Summarise this text: %s", state$get("input_text"))
  }
)

## End(Not run)
```

AgentLogicNode

Agent Logic Node R6 Class

Description

A node that executes a synchronous R function.

Value

An 'AgentLogicNode' object.

Super class

[HydraR::AgentNode](#) -> AgentLogicNode

Public fields

logic_fn Function(state) -> List(status, output).

Methods**Public methods:**

- `AgentLogicNode$new()`
- `AgentLogicNode$run()`
- `AgentLogicNode$clone()`

Method `new()`: Initialize AgentLogicNode

Usage:

```
AgentLogicNode$new(id, logic_fn, label = NULL, params = list())
```

Arguments:

`id` Unique identifier.

`logic_fn` Function that takes an AgentState object and returns a list.

`label` Optional human-readable name.

`params` Optional list of parameters. Run the Logic Node

Method `run()`:

Usage:

```
AgentLogicNode$run(state, ...)
```

Arguments:

`state` AgentState object.

`...` Additional arguments.

Returns: List with status, output, and metadata.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
node <- AgentLogicNode$new("start", function(state) list(status = "success"))
```

AgentMapNode

Agent Map Node R6 Class

Description

A node that maps over a list in the state and performs an operation.

Value

An ‘AgentMapNode’ object.

Super class

HydraR: :AgentNode -> AgentMapNode

Public fields

map_key String. Key in state to map over.

logic_fn Function(item, state) -> List(status, output).

Methods**Public methods:**

- AgentMapNode\$new()
- AgentMapNode\$run()
- AgentMapNode\$clone()

Method new(): Initialize AgentMapNode

Usage:

```
AgentMapNode$new(id, map_key, logic_fn, label = NULL, params = list())
```

Arguments:

id Unique identifier.

map_key String identifier for state retrieval.

logic_fn Mapping function.

label Optional label.

params Optional parameters. Run the Map Node

Method run():

Usage:

```
AgentMapNode$run(state, ...)
```

Arguments:

state AgentState object.

... Additional arguments.

Returns: List with status, output (list of results).

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# Mapping over a list of URLs to fetch data
fetch_logic <- function(url, state) {
  # Custom logic for each item
  list(status = "success", output = paste0("Data from ", url))
}

node_map <- AgentMapNode$new(
  id = "batch_fetcher",
  map_key = "url_list",
  logic_fn = fetch_logic
)

# Setup state with items to map over
state <- AgentState$new(list(url_list = c("url1", "url2", "url3")))
results <- node_map$run(state)

## End(Not run)
```

AgentNode

Agent Node Base Class

Description

The abstract base R6 class for all nodes within an AgentDAG. It defines the common interface and fields required for any node to be orchestrated by the HydraR engine. Subclasses must implement the `run()` method.

Value

An AgentNode object.

Public fields

`id` String. Unique identifier for the node.
`label` String. Human-readable name/label.
`last_result` List. Results from most recent execution.
`params` List. Arbitrary metadata/config parameters.

Methods**Public methods:**

- [AgentNode\\$new\(\)](#)
- [AgentNode\\$run\(\)](#)
- [AgentNode\\$clone\(\)](#)

Method new(): Initialize AgentNode

Usage:

```
AgentNode$new(id, label = NULL, params = list())
```

Arguments:

`id` String. A unique identifier for the node. Must be unique within a single DAG.

`label` String. An optional human-readable name for the node. Defaults to the `id` if not provided. This label is used in Mermaid visualizations.

`params` List. An optional named list of arbitrary metadata or configuration parameters that are stored on the node and can be accessed during execution. Useful for passing static configuration to nodes created via factories. Run the Node

Method run():

Usage:

```
AgentNode$run(state, ...)
```

Arguments:

`state` AgentState. An AgentState object (typically a RestrictedState) providing scoped access to the centralized workflow memory.

`...` Additional arguments. Arbitrary parameters passed from AgentDAG\$run().

Returns: A list with at least `status` (String, e.g., "success", "failed", "pause") and `output` (Any R object to be integrated back into the state).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Defining a custom subclass of AgentNode
CustomNode <- R6::R6Class("CustomNode",
  inherit = AgentNode,
  public = list(
    run = function(state, ...) {
      message("Executing custom node: ", self$id)
      list(status = "success", output = "Custom output")
    }
  )
)

node <- CustomNode$new("node_1", label = "My First Node")
```

AgentObserverNode	<i>Agent Observer Node R6 Class</i>
-------------------	-------------------------------------

Description

A node that executes logic for side-effects (e.g., logging, notifications). Its output does not modify the primary AgentState.

Value

An 'AgentObserverNode' object.

Super class

HydraR: :AgentNode -> AgentObserverNode

Public fields

observe_fn Function(state) -> void.

Methods**Public methods:**

- [AgentObserverNode\\$new\(\)](#)
- [AgentObserverNode\\$run\(\)](#)
- [AgentObserverNode\\$clone\(\)](#)

Method new(): Initialize AgentObserverNode

Usage:

AgentObserverNode\$new(id, observe_fn, label = NULL, params = list())

Arguments:

id Unique identifier.

observe_fn Function that takes an AgentState.

label Optional label.

params Optional parameters. Run the Observer Node

Method run():

Usage:

AgentObserverNode\$run(state, ...)

Arguments:

state AgentState or RestrictedState object.

... Additional arguments.

Returns: List with status "observer" and NULL output.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# An observer that logs stats to a file without modifying state
logger <- function(state) {
  stats <- state$get_all()
  writeLines(jsonlite::toJSON(stats), "logs/observation.json")
  message("Stats observed and written to logs.")
}

node_obs <- AgentObserverNode$new(
  id = "status_logger",
  observe_fn = logger
)

# Run with dummy state
node_obs$run(AgentState$new(list(x = 10, y = 20)))

## End(Not run)
```

AgentPythonNode

Python Execution Node

Description

Executes a python script via system python or reticulate.

Value

An ‘AgentPythonNode’ object.

Super class

[HydraR::AgentNode](#) -> AgentPythonNode

Public fields

`script` Character string or Function. The python script to execute.

`engine` Character. Execution engine ("system2" or "reticulate").

Methods

Public methods:

- [AgentPythonNode\\$new\(\)](#)
- [AgentPythonNode\\$run\(\)](#)
- [AgentPythonNode\\$clone\(\)](#)

Method new():

Usage:

```
AgentPythonNode$new(
  id,
  label = NULL,
  script,
  engine = "system2",
  params = list()
)
```

Arguments:

id Node ID

label Node Label

script String or Function(state) returning a string.

engine "system2" (isolated process) or "reticulate" (inline bindings).

params List of standard Node parameters Run Python Execution

Method run():

Usage:

```
AgentPythonNode$run(state, working_dir = NULL)
```

Arguments:

state AgentState object.

working_dir Directory context for execution.

Returns: List with output, success flag, and optionally result variables.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Pure system-level Python execution (isolated process)
node_sys <- AgentPythonNode$new(
  id = "py_cleaner",
  script = "print('Cleaning data...'); result = 10",
  engine = "system2"
```

```

)

# 2. Reticulate-based execution with shared memory
# This allows the 'result' variable in Python to be returned as an R object.
node_retic <- AgentPythonNode$new(
  id = "py_stats",
  script = "
import numpy as np
data = np.array([1, 2, 3, 4, 5])
result = data.mean()
",
  engine = "reticulate"
)

# Execute and retrieve the 'result' object
res <- node_retic$run(state = AgentState$new())
message("Mean calculated in Python: ", res$result)

## End(Not run)

```

AgentRouterNode

Agent Router Node R6 Class

Description

A node that determines the next node in the DAG dynamically. The logic function must return a list with a 'target_node' field.

Value

An 'AgentRouterNode' object.

Super class

[HydraR::AgentNode](#) -> AgentRouterNode

Public fields

router_fn Function(state) -> List(target_node, output).

Methods

Public methods:

- [AgentRouterNode\\$new\(\)](#)
- [AgentRouterNode\\$run\(\)](#)
- [AgentRouterNode\\$clone\(\)](#)

Method new(): Initialize AgentRouterNode

Usage:

```
AgentRouterNode$new(id, router_fn, label = NULL, params = list())
```

Arguments:

id Unique identifier.

router_fn Function that takes an AgentState and returns a list.

label Optional human-readable name.

params Optional list of parameters. Run the Router Node

Method run():*Usage:*

```
AgentRouterNode$run(state, ...)
```

Arguments:

state AgentState object.

... Additional arguments.

Returns: List with status, output, and target_node.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# Routing based on an LLM sentiment classification
classifier <- function(state) {
  sentiment <- state$get("analyst_output")
  if (grepl("Positive", sentiment)) {
    list(target_node = "celebrate", status = "success")
  } else {
    list(target_node = "investigate", status = "success")
  }
}

node_router <- AgentRouterNode$new(
  id = "sentiment_router",
  router_fn = classifier
)

# Run with dummy state
res <- node_router$run(AgentState$new(list(analyst_output = "Positive results!")))
message("Targeting node: ", res$target_node)

## End(Not run)
```

AgentState

*Agent State R6 Class***Description**

A strongly typed, centrally managed state object for passing data between nodes in an AgentDAG. It supports declarative schemas for validation and functional "reducers" for sophisticated state merging during execution.

Details

The AgentState is the single source of truth for a DAG. It is designed to be serializable, allowing the entire workflow state to be checkpointed and restored.

Value

An AgentState R6 object.

Public fields

`data` Environment. Stores state variables.

`reducers` List. Functions applied to merge updates.

`schema` List. Expected types for state variables.

Methods**Public methods:**

- `AgentState$new()`
- `AgentState$get()`
- `AgentState$get_all()`
- `AgentState$validate()`
- `AgentState$set()`
- `AgentState$update()`
- `AgentState$update_from_node()`
- `AgentState$to_list_serializable()`
- `AgentState$clone()`

Method `new()`: Initialize AgentState

Usage:

```
AgentState$new(initial_data = list(), reducers = list(), schema = list())
```

Arguments:

`initial_data` List or String. The starting data for the state. If a list, it is merged into the environment. If a string, it is stored under the key "input".

reducers Named list of functions or character names. Maps state keys to functions that define how new values are merged with current values (e.g., append or sum).

schema Named list of character strings. Defines the expected class/type for specific keys (e.g., `list(count = "numeric")`). Get a state variable

Method get():

Usage:

`AgentState$get(key, default = NULL)`

Arguments:

`key` String. The name of the variable to retrieve.

`default` Value. The value to return if the key is not found in the state.

Returns: The value associated with the key, or the default. Get all state variables as a list

Method get_all():

Usage:

`AgentState$get_all()`

Returns: A named list. Validate a single state variable against the schema

Method validate():

Usage:

`AgentState$validate(key, value)`

Arguments:

`key` String.

`value` Any value.

Returns: TRUE if valid, throws error otherwise. Set a state variable directly (bypassing reducers)

Method set():

Usage:

`AgentState$set(key, value)`

Arguments:

`key` String.

`value` Any value. Update state using reducers

Method update():

Usage:

`AgentState$update(updates)`

Arguments:

`updates` List of state updates. Update state from a node's output

Method update_from_node():

Usage:

`AgentState$update_from_node(output, node_id)`

Arguments:

output The output from the node.

node_id The ID of the node. Export state for persistence (logic as names)

Method to_list_serializable():*Usage:*

AgentState\$to_list_serializable()

Returns: List.

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

AgentState\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize state with strict schema validation and reducers
state <- AgentState$new(
  initial_data = list(count = 0, logs = list()),
  schema = list(count = "numeric", logs = "list"),
  reducers = list(logs = reducer_append)
)

# 2. Basic update (replaces current value)
state$set("count", 42)

# 3. Reducer update (appends to current list)
state$update(list(logs = "Process started"))
state$update(list(logs = "Processing step 1"))

# 4. Schema validation check
# This will throw an error because 'count' must be numeric
try(state$set("count", "not a number"))

# Verify accumulation
print(state$get("logs")) # list("Process started", "Processing step 1")

## End(Not run)
```

AgentTool

Agent Tool R6 Class

Description

Defines a tool or action that an agent can perform. Used for prompt-based tool discovery.


```

        query = "The DNA sequence string",
        evalue = "The e-value threshold (default 1e-5)"
    ),
    example = "blast_search(query='ATGC...', evalue=0.001)"
)

# Format for injection into a system prompt
message(tool$format())

## End(Not run)

```

AnthropicAPIDriver	<i>Anthropic API Driver</i>
--------------------	-----------------------------

Description

A specialized AgentDriver for the Anthropic Messages API, providing access to the Claude family of models.

Details

Setup: To use this driver, you must set the ANTHROPIC_API_KEY environment variable. It is recommended to add this to your .Renviron file: ANTHROPIC_API_KEY="sk-ant-..."

Value

An AnthropicAPIDriver object.

Super class

[HydraR::AgentDriver](#) -> AnthropicAPIDriver

Public fields

api_url String. Base URL.

Methods

Public methods:

- [AnthropicAPIDriver\\$new\(\)](#)
- [AnthropicAPIDriver\\$get_capabilities\(\)](#)
- [AnthropicAPIDriver\\$call\(\)](#)
- [AnthropicAPIDriver\\$clone\(\)](#)

Method new(): Initialize AnthropicAPIDriver

Usage:

```

AnthropicAPIDriver$new(
  id = "anthropic_api",
  model = "claude-sonnet-4-6",
  validation_mode = "warning",
  working_dir = NULL
)

```

Arguments:

`id` String. Unique identifier for this driver instance.

`model` String. The Anthropic model ID (e.g., "claude-3-sonnet").

`validation_mode` String. Driver validation strictness level.

`working_dir` String. Optional. Base directory for file-system operations.

Returns: A new AnthropicAPIDriver object. Get Capabilities

Method `get_capabilities():`*Usage:*

```
AnthropicAPIDriver$get_capabilities()
```

Returns: A list of capabilities. Call Anthropic API

Method `call():`*Usage:*

```

AnthropicAPIDriver$call(
  prompt,
  model = NULL,
  system_prompt = NULL,
  cli_opts = list(),
  ...
)

```

Arguments:

`prompt` String. The prompt text.

`model` String. Optional model override.

`system_prompt` String. Optional system prompt.

`cli_opts` List. Additional API options.

`...` Additional arguments.

Returns: String. LLM response.

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

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize the Claude Messages API driver
driver <- AnthropicAPIDriver$new(model = "claude-3-opus-20240229")

# 2. Perform a research task with reasoning constraints
response <- driver$call(
  prompt = "Summarize the differences between S3 and R6 classes in R.",
  system_prompt = "You are a technical documentarian. Use markdown tables.",
  cli_opts = list(
    temperature = 0.2,
    max_tokens = 1024,
    stop_sequences = list("### Conclusion")
  )
)
message(response)

## End(Not run)
```

AnthropicCLIDriver	<i>Anthropic CLI Driver</i>
--------------------	-----------------------------

Description

A specialized AgentDriver for the Anthropic claude (Claude Code) CLI. Optimized for terminal-based engineering tasks.

Details

Setup: Requires claude (Anthropic CLI) to be installed and authenticated. Configure the path via `options(HydraR.claude_path = "...")` or in your `.Renviron`.

Value

An AnthropicCLIDriver object.

Super class

`HydraR::AgentDriver` -> AnthropicCLIDriver

Public fields

`model` String. Default model.

Methods

Public methods:

- [AnthropicCLIDriver\\$new\(\)](#)
- [AnthropicCLIDriver\\$call\(\)](#)
- [AnthropicCLIDriver\\$clone\(\)](#)

Method `new()`: Initialize AnthropicCLIDriver

Usage:

```
AnthropicCLIDriver$new(  
  id = "claude_cli",  
  model = "sonnet",  
  validation_mode = "warning",  
  working_dir = NULL  
)
```

Arguments:

`id` Unique identifier.

`model` String. Default model.

`validation_mode` String. "warning" or "strict".

`working_dir` String. Optional. Path to isolated Git worktree.

Returns: A new 'AnthropicCLIDriver' object. Call the LLM

Method `call()`:

Usage:

```
AnthropicCLIDriver$call(  
  prompt,  
  model = NULL,  
  system_prompt = NULL,  
  cli_opts = list(),  
  ...  
)
```

Arguments:

`prompt` String.

`model` String override.

`system_prompt` String. Optional system prompt.

`cli_opts` List.

... Additional arguments.

Returns: String. Cleaned result.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize the Claude Code CLI driver
driver <- AnthropicCLIDriver$new(model = "sonnet")

# 2. Execute a complex engineering task with budget constraints
# and permission skipping for non-interactive automation.
response <- driver$call(
  prompt = "Refactor R/dag.R to use R6 private methods.",
  cli_opts = list(
    dangerously_skip_permissions = TRUE,
    max_budget_usd = 5.0,
    verbose = TRUE
  )
)

## End(Not run)
```

auto_node_factory

Automatic Node Factory for Mermaid-as-Source

Description

Generates a closure that can resolve Mermaid node labels into fully instantiated AgentNode objects based on inline annotations.

Usage

```
auto_node_factory(driver_registry = NULL)
```

Arguments

driver_registry

DriverRegistry. An optional registry used to resolve drivers specified in Mermaid annotations (e.g., driver=openai_api).

Details

The factory supports the following type= parameters in Mermaid labels:

- llm: Creates an AgentLLMNode. Requires role or role_id.
- logic: Creates an AgentLogicNode. Requires logic_id.
- router: Creates an AgentRouterNode. Requires logic_id.
- map: Creates an AgentMapNode. Requires logic_id and map_key.

Value

A function that takes (id, label, params) and returns an AgentNode.

Examples

```
## Not run:
# 'Low Code' workflow: resolve complex attributes from a Mermaid string
mermaid_src <- '
graph TD
  A["Data Fetcher | type=llm | role=Expert Bioinformatician | model=gpt-4o"]
  B["Quality Gate | type=logic | logic_id=validate_data"]
  C["Reporter | type=llm | role=Technical Writer | driver=claude_cli"]

  A --> B
  B --> C
'

# 1. Register the required logic in the registry
register_logic("validate_data", function(state) {
  raw <- state$get("A")
  if (nchar(raw) > 100) list(status = "success") else list(status = "failed")
})

# 2. Create the DAG using the automatic factory
factory <- auto_node_factory()
dag <- mermaid_to_dag(mermaid_src, node_factory = factory)

# Result: n1 is an AgentLLMNode (OpenAI), n2 is Logic, n3 is AgentLLMNode (Claude)

## End(Not run)
```

Checkpointier

Checkpointier Interface

Description

An abstract base class defining the contract for state persistence and recovery in HydraR. Checkpointers allow a DAG to be paused and resumed across sessions by saving the AgentState after each node execution.

Value

A Checkpointer object.

Methods

Public methods:

- [Checkpointier\\$put\(\)](#)
- [Checkpointier\\$get\(\)](#)
- [Checkpointier\\$clone\(\)](#)

Method `put()`: Persist state to the checkpointer.

Usage:

```
Checkpointinter$put(thread_id, state)
```

Arguments:

`thread_id` String. A unique identifier for the execution thread or session.

`state` AgentState. The AgentState object to be persisted.

Returns: NULL (called for side effect). Load state from the checkpointer

Method `get()`:*Usage:*

```
Checkpointinter$get(thread_id)
```

Arguments:

`thread_id` String. The unique identifier associated with the saved state.

Returns: AgentState object or NULL if no checkpoint is found for the thread.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Abstract interface usage (internal)
# Checkpointers are passed to dag_create() to enable state persistence.
dag <- dag_create(
  checkpointer = RDSSaver$new(dir = "checkpoints"),
  message_log = JSONLMessageLog$new(file = "logs/messages.jsonl")
)

## End(Not run)
```

cleanup_jules_branches

Cleanup Stale GitHub Branches

Description

Identifies and optionally deletes stale remote branches on GitHub. Specifically targets branches authored by 'google-labs-jules[bot]' that have been inactive beyond a specified threshold, or any branch already merged into the main branch.

Usage

```
cleanup_jules_branches(
  repo_root = getwd(),
  threshold_hours = 24,
  dry_run = TRUE,
  verbose = TRUE
)
```

Arguments

repo_root	String. Path to the git repository. Defaults to 'getwd()'.
threshold_hours	Numeric. Inactivity threshold for Jules's branches (hours). Defaults to 24.
dry_run	Logical. If TRUE (default), identifies candidates but does not perform deletion.
verbose	Logical. If TRUE, prints detailed status messages.

Value

A character vector of branches identified for deletion.

Examples

```
## Not run:
# 1. Identify stale branches without deleting (Dry Run)
stale_branches <- cleanup_jules_branches(dry_run = TRUE)

# 2. Perform aggressive cleanup of stale bot branches (> 48 hours)
cleanup_jules_branches(
  threshold_hours = 48,
  dry_run = FALSE,
  verbose = TRUE
)

## End(Not run)
```

ConflictResolver

Git Conflict Resolver R6 Class

Description

Handles semantic or git-level conflicts during branch merges. Used by Conflict Harmonizer nodes.

Value

A 'ConflictResolver' R6 object.

Public fields

strategy String. "llm", "human", or "ours".

driver AgentDriver. The driver to use for "llm" strategy.

Methods**Public methods:**

- [ConflictResolver\\$new\(\)](#)
- [ConflictResolver\\$resolve\(\)](#)
- [ConflictResolver\\$clone\(\)](#)

Method new(): Initialize ConflictResolver

Usage:

```
ConflictResolver$new(strategy = "llm", driver = NULL)
```

Arguments:

strategy Conflict resolution strategy.

driver Optional LLM driver for semantic resolution.

Returns: A new 'ConflictResolver' object. Resolve a Conflict

Method resolve():

Usage:

```
ConflictResolver$resolve(repo_root, branch_a, branch_b, files)
```

Arguments:

repo_root Path to repo.

branch_a Branch A.

branch_b Branch B (usually base).

files List of conflicting files.

Returns: List with status and details.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize a resolver with an LLM strategy
resolver <- ConflictResolver$new(
  strategy = "llm",
  driver = AnthropicAPIDriver$new(model = "claude-3-sonnet")
)
```

```
# 2. Resolve a detected conflict during a merge
res <- resolver$resolve(
  repo_root = ".",
  branch_a = "feature-updates",
  branch_b = "main",
  files = c("R/core.R")
)

## End(Not run)
```

CopilotCLIDriver

Copilot CLI Driver R6 Class

Description

A specialized AgentDriver for the GitHub gh copilot CLI extension.

Details

Setup: Requires the GitHub CLI (gh) and the copilot extension (gh extension install github/gh-copilot). Must be authenticated via gh auth login.

Value

A CopilotCLIDriver object.

Super class

`HydraR::AgentDriver` -> CopilotCLIDriver

Methods

Public methods:

- `CopilotCLIDriver$new()`
- `CopilotCLIDriver$call()`
- `CopilotCLIDriver$clone()`

Method `new()`: Initialize CopilotCLIDriver

Usage:

```
CopilotCLIDriver$new(
  id = "copilot_cli",
  validation_mode = "warning",
  working_dir = NULL
)
```

Arguments:

`id` Unique identifier.

`validation_mode` String. "warning" or "strict".

working_dir String. Optional. Path to isolated Git worktree. Call the LLM

Method call():

Usage:

```
CopilotCLIDriver$call(prompt, system_prompt = NULL, cli_opts = list(), ...)
```

Arguments:

prompt String.

system_prompt String. Optional system prompt.

cli_opts List.

... Additional arguments.

Returns: String. Cleaned result.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Use GitHub Copilot CLI extension
driver <- CopilotCLIDriver$new()

# 2. Request a terminal command with tool permissions
response <- driver$call(
  prompt = "Find all CSV files larger than 1MB and move them to 'data/'",
  cli_opts = list(
    allow_all_tools = TRUE,
    no_custom_instructions = FALSE
  )
)

## End(Not run)
```

create_merge_harmonizer

Create a Merge Harmonizer Node

Description

Factory function to create an AgentLogicNode that merges parallel git branches back into the base branch.

Usage

```
create_merge_harmonizer(
  id = "merge_harmonizer",
  strategy = "sequential",
  conflict_resolver = NULL
)
```

Arguments

id	String. Node identifier.
strategy	String. "sequential" or "octopus".
conflict_resolver	ConflictResolver. Optional custom resolver.

Value

An 'AgentLogicNode' R6 object.

Examples

```
## Not run:
# 1. Create a harmonizer with a specific conflict resolution strategy
resolver <- ConflictResolver$new(strategy = "human")
merger <- create_merge_harmonizer(
  id = "global_merger",
  strategy = "sequential",
  conflict_resolver = resolver
)

# 2. This node is then added to a DAG that uses parallel execution
# and a WorktreeManager in its state.

## End(Not run)
```

dag_add_llm_node	<i>Add an LLM Agent Node directly to a DAG</i>
------------------	--

Description

Instantiates an AgentLLMNode and appends it to the provided AgentDAG in one step.

Usage

```
dag_add_llm_node(dag, id, role, driver, model = NULL, cli_opts = list(), ...)
```

Arguments

dag	AgentDAG. The graph object to which the node will be added.
id	String. Unique identifier for the node.
role	String. System prompt/role for the agent.
driver	AgentDriver. The LLM driver instance.
model	String. Optional model name override.
cli_opts	List. Optional CLI/API parameters.
...	Additional arguments. Passed to AgentLLMNode\$new().

Value

The modified AgentDAG object (invisibly).

Examples

```
## Not run:
# Chaining nodes using the functional API
# This pattern is common in complex HydraR orchestration scripts.
dag <- dag_create() |>
  dag_add_llm_node(
    id = "writer",
    role = "Academic technical writer",
    driver = OpenAIAPIDriver$new(api_key = Sys.getenv("OPENAI_API_KEY"), model = "gpt-4-turbo")
  ) |>
  dag_add_llm_node(
    id = "critic",
    role = "Peer reviewer",
    driver = AnthropicAPIDriver$new(api_key = Sys.getenv("ANTHROPIC_API_KEY"), model = "claude-3-sonnet"),
    cli_opts = list(temperature = 0.2)
  )

## End(Not run)
```

dag_add_logic_node	<i>Add an R Logic Node directly to a DAG</i>
--------------------	--

Description

Add an R Logic Node directly to a DAG

Usage

```
dag_add_logic_node(dag, id, logic_fn, ...)
```

Arguments

dag	AgentDAG object.
id	String. Unique identifier for the node.
logic_fn	Function. Pure R function taking an AgentState object.
...	Additional arguments passed to AgentLogicNode\$new()

Value

The modified AgentDAG object (invisibly).

Examples

```
## Not run:
# Adding logic nodes to a production pipeline
dag <- dag_create() |>
  dag_add_llm_node("planner", "Project Manager", GeminiCLIDriver$new()) |>
  dag_add_logic_node("audit_check", function(state) {
    # Log state for external observability
    message("Auditing current progress...")
    list(status = "success", output = list(timestamp = Sys.time()))
  })

## End(Not run)
```

dag_create

Create an Agent Graph

Description

Initializes a new AgentDAG object. This is the primary entry point for building orchestration workflows either programmatically or from definitions.

Usage

```
dag_create(message_log = NULL)
```

Arguments

message_log	MessageLog. An optional MessageLog R6 object (e.g., MemoryMessageLog or DuckDBMessageLog) used to capture all inter-node communication for auditing and debugging.
-------------	--

Value

An AgentDAG R6 object.

Examples

```
## Not run:
library(HydraR)

# 1. Simple Audit-only DAG
dag <- dag_create()

# 2. Production DAG with persistent DuckDB audit log and custom metadata
# This pattern ensures every agent interaction is recorded for reproducibility.
log <- DuckDBMessageLog$new(db_path = "audit_trail.duckdb")
dag <- dag_create(message_log = log)

# 3. Advanced setup with a pre-configured memory saver
saver <- MemorySaver$new()
cp <- Checkpointer$new(saver = saver)
# Note: Checkpointer is typically assigned to the DAG after creation
dag$checkpointer <- cp

## End(Not run)
```

DriverRegistry

Driver Registry R6 Class

Description

A singleton-like registry for managing AgentDriver instances. Enables runtime recovery and hot-swapping of drivers across nodes.

Value

A ‘DriverRegistry’ R6 object.

Public fields

`drivers` List. Storage for registered drivers.

Methods**Public methods:**

- `DriverRegistry$new()`
- `DriverRegistry$register()`
- `DriverRegistry$get()`
- `DriverRegistry$list_drivers()`
- `DriverRegistry$remove()`
- `DriverRegistry$clear()`
- `DriverRegistry$clone()`

Method new(): Initialize DriverRegistry

Usage:

DriverRegistry\$new()

Returns: A new 'DriverRegistry' object. Register a Driver

Method register():

Usage:

DriverRegistry\$register(driver, overwrite = FALSE)

Arguments:

driver AgentDriver object.

overwrite Logical. Whether to overwrite existing driver with same ID.

Returns: The registry object (invisibly). Get a Driver

Method get():

Usage:

DriverRegistry\$get(id)

Arguments:

id String. Driver identifier.

Returns: AgentDriver object or NULL. List Registered Drivers

Method list_drivers():

Usage:

DriverRegistry\$list_drivers()

Returns: Data frame of registered drivers and their metadata. Remove a Driver

Method remove():

Usage:

DriverRegistry\$remove(id)

Arguments:

id String.

Returns: The registry object (invisibly). Clear Registry

Method clear():

Usage:

DriverRegistry\$clear()

Returns: The registry object (invisibly).

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

Usage:

DriverRegistry\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Create a registry and register multiple models
reg <- DriverRegistry$new()
reg$register(GeminiCLIDriver$new(id = "fast_model", model = "gemini-1.5-flash"))
reg$register(GeminiCLIDriver$new(id = "smart_model", model = "gemini-1.5-pro"))

# 2. Audit the registered drivers
summary <- reg$list_drivers()
print(summary)

# 3. Fetch a specific driver by its shorthand ID
my_drv <- reg$get("smart_model")

## End(Not run)
```

DuckDBMessageLog

*DuckDB Message Log R6 Class***Description**

A persistent implementation of MessageLog that writes messages to a centralized DuckDB database. This is the recommended logger for production and audit-heavy workflows.

Value

A DuckDBMessageLog object.

Super class

[HydraR: :MessageLog](#) -> DuckDBMessageLog

Public fields

db_path String. Path to DuckDB file.

Methods**Public methods:**

- [DuckDBMessageLog\\$new\(\)](#)
- [DuckDBMessageLog\\$log\(\)](#)
- [DuckDBMessageLog\\$get_all\(\)](#)
- [DuckDBMessageLog\\$clone\(\)](#)

Method `new()`: Finalizer to clean up the cached connection.

Initialize DuckDBMessageLog.

Usage:

```
DuckDBMessageLog$new(db_path = "~/gemini/memory/bot_history.duckdb")
```

Arguments:

`db_path` String. Path to the DuckDB file. If the file does not exist, it will be created upon the first message log. Defaults to the master `bot_history.duckdb`.

Returns: A new DuckDBMessageLog object.

Method `log()`: Store a message.

Usage:

```
DuckDBMessageLog$log(msg)
```

Arguments:

`msg` List. Message object.

Returns: The log object (invisibly).

Method `get_all()`: Get all logs.

Usage:

```
DuckDBMessageLog$get_all()
```

Returns: List of logs.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Persistent audit logging to DuckDB
audit_log <- DuckDBMessageLog$new(
  db_path = "data/agent_audit.duckdb"
)

# 2. Attach to a DAG and run
dag <- dag_create(message_log = audit_log)
dag$run(thread_id = "experiment_404")

# 3. Retrieve and analyze messages from a previous session
prev_logs <- audit_log$get_all()

## End(Not run)
```

DuckDBSaver

*DuckDBSaver Checkpointer***Description**

A production-grade Checkpointer that utilizes DuckDB for high-performance state persistence. Supports BLOB storage of serialized R objects and concurrent access patterns.

Value

A DuckDBSaver R6 object.

Super class

[HydraR::Checkpoint](#) -> DuckDBSaver

Public fields

con DBIConnection.

table_name String.

Methods**Public methods:**

- [DuckDBSaver\\$new\(\)](#)
- [DuckDBSaver\\$put\(\)](#)
- [DuckDBSaver\\$get\(\)](#)
- [DuckDBSaver\\$clone\(\)](#)

Method new(): Initialize DuckDBSaver

Usage:

```
DuckDBSaver$new(con = NULL, db_path = NULL, table_name = "agent_checkpoints")
```

Arguments:

con DBIConnection. An existing DBI connection to a DuckDB instance.

db_path String. Path to a DuckDB file. If provided, the driver will handle the connection internally.

table_name String. The name of the table used to store checkpoints. Defaults to "agent_checkpoints".
Save state

Method put():

Usage:

```
DuckDBSaver$put(thread_id, state)
```

Arguments:

thread_id String.

state AgentState object. Load state

Method get():

Usage:

DuckDBSaver\$get(thread_id)

Arguments:

thread_id String.

Returns: AgentState object or NULL.

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

Usage:

DuckDBSaver\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Production-ready persistence using DuckDB
saver <- DuckDBSaver$new(
  db_path = "storage/hydrar_main.duckdb",
  table_name = "workflow_checkpoints"
)

# 2. Orchestrate a long-running DAG
dag <- dag_create(checkpointer = saver)
dag$run(thread_id = "genomic_alignment_job_42")

# 3. Query the internal storage directly if needed
library(DBI)
DBI::dbGetQuery(saver$con, "SELECT thread_id, updated_at FROM workflow_checkpoints")

## End(Not run)
```

extract_r_code_advanced
<i>Extract R Code from LLM Response</i>

Description

Extract R Code from LLM Response

Usage

extract_r_code_advanced(raw)

Arguments

raw String. Raw text response from LLM.

Value

String. Extracted R code or same text if no blocks found.

Examples

```
## Not run:
# 1. Extract a single R block from markdown noise
raw_text <- "Here is the code: \n```r\nplot(cars)\n```\nHope it helps!"
extract_r_code_advanced(raw_text)

# 2. Extract and concatenate multiple blocks
multi_blocks <- "First: ```R\nx <- 1\n``` Then: ```r\ny <- x + 1\n```"
code <- extract_r_code_advanced(multi_blocks)
# Result: \n"x <- 1\n\ny <- x + 1\n"

## End(Not run)
```

format_toolset	<i>Format Toolset for Prompt</i>
----------------	----------------------------------

Description

Format Toolset for Prompt

Usage

```
format_toolset(tools)
```

Arguments

tools List of AgentTool objects.

Value

A formatted string containing all tool descriptions.

Examples

```
## Not run:
# Format a collection of tools for an agent
tools <- list(
  search = AgentTool$new("google_search", "Search the web"),
  run_r = AgentTool$new("r_exec", "Execute R code locally")
)
```

```
prompt_appendix <- format_toolset(tools)
cat(prompt_appendix)

## End(Not run)
```

GeminiAPIDriver

Gemini API Driver R6 Class

Description

A specialized AgentDriver for the Google Gemini API (AI Studio / Vertex AI).

Details

Setup: Requires the GOOGLE_API_KEY environment variable to be set. Add it to your .Renviron: GOOGLE_API_KEY="AIza..."

Value

A GeminiAPIDriver R6 object.

Super class

[HydraR::AgentDriver](#) -> GeminiAPIDriver

Public fields

api_base String. Base URL.

Methods

Public methods:

- [GeminiAPIDriver\\$new\(\)](#)
- [GeminiAPIDriver\\$get_capabilities\(\)](#)
- [GeminiAPIDriver\\$call\(\)](#)
- [GeminiAPIDriver\\$clone\(\)](#)

Method new(): Initialize GeminiAPIDriver

Usage:

```
GeminiAPIDriver$new(
  id = "gemini_api",
  model = "gemini-3.1-flash-lite-preview",
  validation_mode = "warning",
  working_dir = NULL
)
```

Arguments:

id String. Unique identifier for this driver.

model String. Google model ID (e.g., "gemini-1.5-flash").
 validation_mode String. Controls schema enforcement behavior.
 working_dir String. Optional path for isolated execution.
Returns: A new GeminiAPIDriver object. Get Capabilities

Method get_capabilities():

Usage:
 GeminiAPIDriver\$get_capabilities()
Returns: A list of capabilities supported by the Gemini API. Call Gemini API

Method call():

Usage:
 GeminiAPIDriver\$call(
 prompt,
 model = NULL,
 system_prompt = NULL,
 cli_opts = list(),
 ...
)
Arguments:
 prompt String. The prompt or query provided by the user.
 model String. Optional override for the model ID.
 system_prompt String. Optional system-level instructions for the model.
 cli_opts List. Advanced API parameters (e.g., generationConfig).
 ... Additional arguments passed to the requester.
Returns: String. The text response generated by Gemini.

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

Usage:
 GeminiAPIDriver\$clone(deep = FALSE)
Arguments:
 deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize Gemini Pro API
driver <- GeminiAPIDriver$new(model = "gemini-1.5-pro")

# 2. Advanced call with custom generation config and safety filtering
response <- driver$call(
  prompt = "Write a high-performance R function for matrix multiplication.",
  cli_opts = list(
    generationConfig = list(
      temperature = 0.9,
      topP = 0.8,
```

```

        topK = 40,
        maxOutputTokens = 2048
    ),
    safetySettings = list(
        list(category = "HARM_CATEGORY_HARASSMENT", threshold = "BLOCK_LOW_AND_ABOVE")
    )
)
)

## End(Not run)

```

GeminiCLIDriver

Gemini CLI Driver R6 Class

Description

A specialized AgentDriver that invokes the Google gemini CLI tool. This is the preferred driver for workflows requiring local tool use and filesystem interaction via the Google-native MCP bridge.

Details

Setup: Requires the gemini CLI to be installed and in your PATH. You can override the binary path using: `options(HydraR.gemini_path = "/path/to/gemini")` or by setting the `HYDRAR_GEMINI_PATH` environment variable in your `.Renv`.

Value

A GeminiCLIDriver object.

Super class

`HydraR::AgentDriver` -> GeminiCLIDriver

Public fields

`model` String. Default model. Omit to use CLI default.

Methods

Public methods:

- `GeminiCLIDriver$new()`
- `GeminiCLIDriver$call()`
- `GeminiCLIDriver$clone()`

Method `new()`: Initialize GeminiCLIDriver

Usage:

```
GeminiCLIDriver$new(
  id = "gemini_cli",
  model = "gemini-2.5-flash",
  validation_mode = "warning",
  working_dir = NULL,
  repo_root = NULL
)
```

Arguments:

`id` String. Unique identifier for this driver.
`model` String. The Gemini model ID (e.g., "gemini-1.5-flash").
`validation_mode` String. Either "warning" or "strict".
`working_dir` String. Optional. Path to an isolated git worktree where the CLI will execute.
`repo_root` String. Optional. Path to the main repository to enable cross-worktree context. Call the LLM

Method `call()`:

Usage:

```
GeminiCLIDriver$call(
  prompt,
  model = NULL,
  system_prompt = NULL,
  cli_opts = list(),
  ...
)
```

Arguments:

`prompt` String.
`model` String override.
`system_prompt` String. Optional system prompt.
`cli_opts` List. Named list of CLI options.
`...` Additional arguments.

Returns: String. Cleaned result.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Standard CLI-based agent with model selection
driver <- GeminiCLIDriver$new(model = "gemini-1.5-pro")

# 2. Advanced call with MCP tool discovery and 'YOLO' mode enabled
```

```
# This allows the agent to execute tools without interactive confirmation.
response <- driver$call(
  prompt = "Analyze the R scripts in this directory and suggest performance fixes.",
  cli_opts = list(
    allowed_tools = "ls,grep,read_file",
    allowed_mcp_server_names = "filesystem,github",
    yolo = TRUE,
    include_directories = c("R", "tests")
  )
)

## End(Not run)
```

GeminiImageDriver

*Gemini Image API Driver R6 Class***Description**

A specialized driver for Gemini's multimodal and image generation capabilities (supporting Imagen and native Gemini 3.x modalities).

Details

Setup: Requires GOOGLE_API_KEY in your .Renviron.

Value

A GeminiImageDriver R6 object.

Super classes

[HydraR::AgentDriver](#) -> [HydraR::GeminiAPIDriver](#) -> GeminiImageDriver

Public fields

output_dir String. Directory to save generated images.

aspect_ratio String. Default "16:9".

Methods**Public methods:**

- [GeminiImageDriver\\$new\(\)](#)
- [GeminiImageDriver\\$call\(\)](#)
- [GeminiImageDriver\\$clone\(\)](#)

Method new(): Initialize GeminiImageDriver

Usage:

```
GeminiImageDriver$new(
  id = "gemini_image",
  model = "gemini-3.1-flash-image-preview",
  output_dir = "images",
  aspect_ratio = "1:1",
  validation_mode = "warning",
  working_dir = NULL
)
```

Arguments:

`id` String. Unique identifier for the image driver.
`model` String. Google model ID (defaults to multimodal flash).
`output_dir` String. The directory where generated images will be saved.
`aspect_ratio` String. The default aspect ratio for generated images (e.g., "16:9", "1:1").
`validation_mode` String. Controls schema enforcement.
`working_dir` String. Optional path for isolated execution.

Returns: A new GeminiImageDriver instance. Call Gemini Image API (Multimodal Unified)

Method `call()`: Sends a prompt to Gemini for image generation.

Usage:

```
GeminiImageDriver$call(
  prompt,
  model = NULL,
  system_prompt = NULL,
  cli_opts = list(),
  ...
)
```

Arguments:

`prompt` String. Image prompt.
`model` String. Optional override.
`system_prompt` String. Optional system prompt.
`cli_opts` List. Parameters (aspectRatio, etc).
`...` Additional arguments.

Returns: String. Local path to the generated image.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Generate a high-resolution laboratory illustration
```

```
# Multimodal Gemini 3.1 models infer dimensions from prompt + config
driver <- GeminiImageDriver$new(output_dir = "assets/media")

# 2. Request a specific aspect ratio and filename
img_path <- driver$call(
  prompt = "A futuristic bioinformatics lab with DNA holograms, hyper-realistic, 8k",
  cli_opts = list(
    aspectRatio = "16:9",
    sampleCount = 1,
    filename = "hero_dna_lab.png"
  )
)
message("Hero image saved to: ", img_path)

## End(Not run)
```

get_agent_roles	<i>List Registered Roles</i>
-----------------	------------------------------

Description

List Registered Roles

Usage

```
get_agent_roles()
```

Value

Character vector of role names.

Examples

```
## Not run:
# List all personas currently registered in the system
personas <- get_agent_roles()
stopifnot("bio_analyst" %in% personas)

## End(Not run)
```

get_default_driver	<i>Get the Default Agent Driver</i>
--------------------	-------------------------------------

Description

Get the Default Agent Driver

Usage

```
get_default_driver()
```

Value

AgentDriver object or NULL.

Examples

```
## Not run:  
# Resolve the driver to be used when none is explicitly specified  
drv <- get_default_driver()  
if (!is.null(drv)) {  
  message("Using default provider: ", drv$provider)  
}  
  
## End(Not run)
```

get_driver_registry	<i>Global Driver Registry Accessor</i>
---------------------	--

Description

Global Driver Registry Accessor

Usage

```
get_driver_registry()
```

Value

The global DriverRegistry instance.

Examples

```
## Not run:
# Access the singleton registry used across the entire package
reg <- get_driver_registry()

# Register a global driver that any node can reference by name 'default'
reg$register(OpenAIAPIDriver$new(id = "default"), overwrite = TRUE)

## End(Not run)
```

`get_logic`*Get Logic Function*

Description

Get Logic Function

Usage

```
get_logic(name)
```

Arguments

name	String. Unique identifier.
------	----------------------------

Value

Function or NULL.

Examples

```
## Not run:
# Retrieve a function by name for manual node construction
logic_fn <- get_logic("validate_results")
if (!is.null(logic_fn)) {
  node <- AgentLogicNode$new(id = "validator", logic_fn = logic_fn)
}

## End(Not run)
```

get_role	<i>Get an LLM Role (System Prompt)</i>
----------	--

Description

Get an LLM Role (System Prompt)

Usage

get_role(name)

Arguments

name String. Unique identifier.

Value

String prompt text, or NULL if not found.

Examples

```
## Not run:
# Resolve a role for an LLM node manually
role_prompt <- get_role("bio_analyst")
node <- AgentLLMNode$new(id = "a1", role = role_prompt, driver = GeminiCLIDriver$new())

## End(Not run)
```

get_role_prompt	<i>Get a Role-specific System Prompt</i>
-----------------	--

Description

Get a Role-specific System Prompt

Usage

get_role_prompt(name)

Arguments

name String. Role identifier.

Value

String prompt text.

Examples

```
## Not run:
# Convenience helper to get role prompts from the Logic Registry
register_role("assistant", "You are a helpful assistant.")
p <- get_role_prompt("assistant")

## End(Not run)
```

init_bot_history	<i>Initialize Bot History DuckDB</i>
------------------	--------------------------------------

Description

Connects to the master DuckDB for telemetry and logs. Follows APAF Bioinformatics mandatory pattern for database persistence.

Usage

```
init_bot_history(read_only = FALSE)
```

Arguments

read_only Logical. If TRUE, connects in read-only mode.

Value

A DBIConnection object.

Examples

```
## Not run:
# Connect to the master bot history for auditing
con <- init_bot_history(read_only = TRUE)

# Use DBI to query the execution logs
library(DBI)
dbListTables(con)
dbDisconnect(con)

## End(Not run)
```

is_named_list	<i>Check if an object is a named list</i>
---------------	---

Description

Helper to verify that an object is a list (but not a data.frame) and that all its elements have non-empty names.

Usage

```
is_named_list(x)
```

Arguments

x Any object.

Value

Logical.

Examples

```
## Not run:
# Valid named list
is_named_list(list(a = 1, b = 2)) # TRUE

# Invalid: contains unnamed elements
is_named_list(list(a = 1, 2)) # FALSE

# Invalid: data.frame (technically a list, but we want a record/dict)
is_named_list(data.frame(a = 1)) # FALSE

## End(Not run)
```

JSONLMessageLog	<i>JSONL Message Log R6 Class</i>
-----------------	-----------------------------------

Description

A file-based implementation of MessageLog that appends messages to a JSON Lines file. This implementation is safe for parallel execution across git worktrees as it uses atomic line appending.

Value

A JSONLMessageLog object.

Super class

HydraR: :MessageLog -> JSONLMessageLog

Public fields

path String. Path to JSONL file.

Methods**Public methods:**

- JSONLMessageLog\$new()
- JSONLMessageLog\$log()
- JSONLMessageLog\$get_all()
- JSONLMessageLog\$clone()

Method new(): Initialize JSONLMessageLog.

Usage:

```
JSONLMessageLog$new(path = tempfile(fileext = ".jsonl"))
```

Arguments:

path String. The output file path for the JSON Lines log. Defaults to a temporary file.

Method log(): Store a message (atomic append).

Usage:

```
JSONLMessageLog$log(msg)
```

Arguments:

msg List. Message object.

Method get_all(): Get all logs.

Usage:

```
JSONLMessageLog$get_all()
```

Returns: List of logs.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Create a file-based logger using JSONL format
file_log <- JSONLMessageLog$new(path = "logs/pipeline_audit.jsonl")

# 2. Orchestrate a DAG with file-level auditing
dag <- dag_create(message_log = file_log)
dag$run(initial_state = list(x = 1))

# 3. Read back the logs from the file
logs <- file_log$get_all()
print(logs[[1]])

## End(Not run)
```

list_logic	List Registered Components
------------	----------------------------

Description

List Registered Components

Usage

```
list_logic()
```

Value

Character vector of names.

Examples

```
## Not run:
# Check which custom logic functions are currently available
available_logic <- list_logic()
message("Registered logic: ", paste(available_logic, collapse = ", "))

## End(Not run)
```

load_workflow	<i>Load Multi-Agent Workflow from File</i>
---------------	--

Description

Parses a declarative workflow definition from a YAML or JSON file. This function handles the resolution of external logic files and role-playing identities referenced in the definition.

Usage

```
load_workflow(file_path)
```

Arguments

`file_path` String. The absolute or relative path to a .yaml, .yml, or .json file.

Value

A list representing the workflow structure, including graph, initial_state, roles, and logic mapping.

Examples

```
## Not run:
# 1. Prepare a YAML workflow with relative logic paths
# my_workflow.yaml:
# roles:
#   analyst: "You are an analyst."
# graph: |
#   graph TD
#     A[type=llm | role_id=analyst]-->B[type=logic | logic_id=clean_data]
# logic:
#   clean_data: "scripts/clean.R"

wf <- load_workflow("my_workflow.yaml")

# 2. Key components are resolved during loading
print(wf$roles$analyst)
print(wf$logic$clean_data) # Returns the function sourced from scripts/clean.R

## End(Not run)
```

MemoryMessageLog	<i>Memory Message Log R6 Class</i>
------------------	------------------------------------

Description

A non-persistent, in-memory implementation of MessageLog. Messages are stored in an internal list that exists only for the duration of the R session.

Value

A MemoryMessageLog object.

Super class

[HydraR::MessageLog](#) -> MemoryMessageLog

Public fields

logs List. Storage for message logs.

Methods**Public methods:**

- [MemoryMessageLog\\$log\(\)](#)
- [MemoryMessageLog\\$get_all\(\)](#)
- [MemoryMessageLog\\$clone\(\)](#)

Method `log()`: Store a message.

Usage:

`MemoryMessageLog$log(msg)`

Arguments:

msg List. Message object.

Returns: The log object (invisibly).

Method `get_all()`: Get all logs.

Usage:

`MemoryMessageLog$get_all()`

Returns: List of logs.

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

Usage:

`MemoryMessageLog$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Create a new memory logger for rapid prototyping
log <- MemoryMessageLog$new()

# 2. Run a DAG and inspect logs after execution
dag <- dag_create(message_log = log)
dag$run(initial_state = list(input = "Hello"))

# Inspect total messages captured
all_messages <- log$get_all()
message("Captured ", length(all_messages), " messages.")

## End(Not run)
```

MemorySaver

MemorySaver Checkpointer

Description

An in-memory implementation of the Checkpointer interface. Stores checkpoints in a dedicated R environment. This is useful for testing or short-lived sessions where persistence to disk is not required.

Value

A MemorySaver object.

Super class

[HydraR::Checkpointer](#) -> MemorySaver

Public fields

storage Environment. Stores the states.

Methods**Public methods:**

- [MemorySaver\\$new\(\)](#)
- [MemorySaver\\$put\(\)](#)
- [MemorySaver\\$get\(\)](#)
- [MemorySaver\\$clone\(\)](#)

Method [new\(\)](#): Initialize MemorySaver

Creates a new environment for in-memory checkpoint storage.

Usage:

```
MemorySaver$new()
```

Method `put()`: Persist state to the checkpointer.

Usage:

```
MemorySaver$put(thread_id, state)
```

Arguments:

`thread_id` String.

`state` AgentState object.

Method `get()`: Load state from the checkpointer.

Usage:

```
MemorySaver$get(thread_id)
```

Arguments:

`thread_id` String.

Returns: AgentState object or NULL.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Use MemorySaver for transient testing or short sessions
saver <- MemorySaver$new()
dag <- dag_create(checkpointer = saver)

# 2. States are saved in the 'storage' environment under thread_id
dag$run(thread_id = "test_run_01")
ls(saver$storage) # Returns "test_run_01"

## End(Not run)
```

mermaid_to_dag

Create AgentDAG from Mermaid

Description

Create AgentDAG from Mermaid

Usage

```
mermaid_to_dag(mermaid_str, node_factory = auto_node_factory())
```

Arguments

`mermaid_str` String. Mermaid syntax.

`node_factory` Function(id, label) -> AgentNode. Defaults to 'auto_node_factory()'.

Value

The AgentDAG object.

Examples

```
## Not run:
# 1. High-level 'Logic-First' graph
# Mapping Mermaid node labels to registered R functions
dag1 <- mermaid_to_dag("graph TD; A[logic:fetch]-->B[logic:validate];")

# 2. Complex 'Agent-First' graph with extended metadata
# Explicitly defining roles, drivers, and models inside attributes
mermaid_src <- '
graph TD
  A["Researcher | type=llm | role=Expert Analyst | model=sonnet"]
  B["Critic | type=llm | role=Scientific Reviewer | temp=0.2"]
  C["Refiner | type=logic | logic_id=refine_markdown"]

  A --> B
  B -- (feedback_required) --> A
  B -- (approved) --> C
,
```

```
# Spawning the DAG
dag2 <- mermaid_to_dag(mermaid_src)

# Verification
print(dag2$nodes$A$role) # "Expert Analyst"

## End(Not run)
```

MessageLog

Message Log Base R6 Class

Description

An abstract base class defining the interface for message logging in HydraR. Subclasses provide concrete storage implementations (Memory, File, Database).

Value

A MessageLog base object.

Methods

Public methods:

- `MessageLog$log()`
- `MessageLog$get_all()`
- `MessageLog$clone()`

Method `log()`: Store a message.

Usage:

`MessageLog$log(msg)`

Arguments:

`msg` List. Message object.

Returns: The log object (invisibly).

Method `get_all()`: Get all logs.

Usage:

`MessageLog$get_all()`

Returns: List of logs.

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

Usage:

`MessageLog$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Abstract interface usage (internal)
# Message logs are passed to dag_create() for audit trails.
dag <- dag_create(
  message_log = DuckDBMessageLog$new(db_path = "audit.duckdb")
)

## End(Not run)
```

OllamaDriver

Ollama Driver R6 Class

Description

A specialized AgentDriver for local execution of Open source models via the ollama CLI. Ideal for air-gapped or privacy-sensitive workflows.

Details

Setup: Ensure the ollama server is running locally. You can specify the binary path via options(`HydraR.ollama_path = "..."`).

Value

An OllamaDriver object.

Super class

`HydraR::AgentDriver -> OllamaDriver`

Public fields

`model` String. Default model.

Methods**Public methods:**

- `OllamaDriver$new()`
- `OllamaDriver$format_cli_opts()`
- `OllamaDriver$call()`
- `OllamaDriver$clone()`

Method `new()`: Initialize OllamaDriver

Usage:

```
OllamaDriver$new(  
  id = "ollama",  
  model = "llama3.2",  
  validation_mode = "warning",  
  working_dir = NULL  
)
```

Arguments:

`id` Unique identifier.

`model` String. Default model.

`validation_mode` String. "warning" or "strict".

`working_dir` String. Optional. Path to isolated Git worktree. Format CLI Options for Ollama

Method `format_cli_opts()`:

Usage:

```
OllamaDriver$format_cli_opts(cli_opts = list())
```

Arguments:

`cli_opts` List.

Returns: Character vector. Call the LLM

Method `call()`:

Usage:

```
OllamaDriver$call(
  prompt,
  model = NULL,
  system_prompt = NULL,
  cli_opts = list(),
  ...
)
```

Arguments:

prompt String.
 model String override.
 system_prompt String. Optional system prompt.
 cli_opts List.
 ... Additional arguments.

Returns: String. Cleaned result.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Use llama3.2 locally via Ollama
driver <- OllamaDriver$new(model = "llama3.2")

# 2. Call with high-precision sampling and large context window
# Ideal for summarizing long bioinformatics documents.
response <- driver$call(
  prompt = "Summarize the technical specifications for the 'samtools' format.",
  cli_opts = list(
    num_ctx = 32768,
    temperature = 0.1,
    repeat_penalty = 1.2,
    seed = 42
  )
)

## End(Not run)
```

OpenAIAPIDriver

OpenAI API Driver

Description

A specialized AgentDriver that interacts with the OpenAI Chat Completions API. Requires an active OpenAI account and API key.

Details

Setup: To use this driver, you must set the OPENAI_API_KEY environment variable. It is recommended to add this to your .Renviron file: OPENAI_API_KEY="sk-..."

Value

An OpenAIAPIDriver object.

Super class

HydraR::AgentDriver -> OpenAIAPIDriver

Public fields

api_url String. Base URL.

Methods**Public methods:**

- OpenAIAPIDriver\$new()
- OpenAIAPIDriver\$get_capabilities()
- OpenAIAPIDriver\$call()
- OpenAIAPIDriver\$clone()

Method new(): Initialize OpenAIAPIDriver

Usage:

```
OpenAIAPIDriver$new(
  id = "openai_api",
  model = "gpt-5.4-mini",
  validation_mode = "warning",
  working_dir = NULL
)
```

Arguments:

id String. Unique identifier for this driver instance.

model String. The OpenAI model ID (e.g., "gpt-4", "gpt-3.5-turbo").

validation_mode String. Either "warning" or "strict". Controls how schema mismatches are handled.

`working_dir` String. Optional. The directory where output files should be generated.

Returns: A new OpenAIAPIDriver object. Get Capabilities

Method `get_capabilities()`:

Usage:

```
OpenAIAPIDriver$get_capabilities()
```

Returns: A list of capabilities. Call OpenAI API

Method `call()`:

Usage:

```
OpenAIAPIDriver$call(
  prompt,
  model = NULL,
  system_prompt = NULL,
  cli_opts = list(),
  ...
)
```

Arguments:

`prompt` String. The primary user message or instruction sent to the LLM.

`model` String. Optional model override for this specific call.

`system_prompt` String. Optional system-level instruction (role-playing or constraint).

`cli_opts` List. Additional parameters passed to the JSON body (e.g., `temperature = 0.7`).

`...` Additional arguments. Passed through to the internal request handler.

Returns: String. The text content of the LLM's response.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Standard API-based completion
driver <- OpenAIAPIDriver$new(model = "gpt-4-turbo")

# 2. Advanced call with JSON mode and deterministic sampling
# This ensures the LLM returns a strictly formatted JSON string.
response <- driver$call(
  prompt = "Return a JSON list of 3 common R packages for plotting.",
  cli_opts = list(
    temperature = 0,
    response_format = list(type = "json_object"),
    max_tokens = 500
  )
)
```

```

)
message(response)

## End(Not run)

```

OpenAICodexCLIDriver *OpenAI Codex CLI Driver R6 Class*

Description

A specialized AgentDriver for the official codex CLI tool (v0.118+). This model is legacy but still supported for specific code-generation tasks.

Value

An OpenAICodexCLIDriver object.

Super class

[HydraR::AgentDriver](#) -> OpenAICodexCLIDriver

Methods

Public methods:

- [OpenAICodexCLIDriver\\$new\(\)](#)
- [OpenAICodexCLIDriver\\$call\(\)](#)
- [OpenAICodexCLIDriver\\$clone\(\)](#)

Method `new()`: Initialize OpenAICodexCLIDriver

Usage:

```

OpenAICodexCLIDriver$new(
  id = "codex_cli",
  validation_mode = "warning",
  working_dir = NULL
)

```

Arguments:

`id` String. Unique identifier.

`validation_mode` String. "warning" or "strict".

`working_dir` String. Optional. Path to isolated Git worktree. Call the LLM

Method `call()`: Sends a execution request to the Codex CLI.

Usage:

```

OpenAICodexCLIDriver$call(prompt, system_prompt = NULL, cli_opts = list(), ...)

```

Arguments:

`prompt` String.

system_prompt String. Optional system prompt.
 cli_opts List.
 ... Additional arguments.

Returns: String. Cleaned result.

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

Usage:

OpenAICodexCLIDriver\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Legacy Codex CLI support
driver <- OpenAICodexCLIDriver$new()

# 2. Isolated execution with skip-checks for specific environments
response <- driver$call(
  prompt = "Create a data.frame with 5 rows and 2 columns.",
  cli_opts = list(
    sandbox = TRUE,
    skip_git_repo_check = TRUE
  )
)

## End(Not run)
```

parse_mermaid

Parse Mermaid Flowchart Syntax

Description

Extracts nodes and edges from 'graph TD' or 'flowchart TD' strings. Supports basic node labels and directed edges.

Usage

```
parse_mermaid(mermaid_str)
```

Arguments

mermaid_str String. Mermaid syntax.

Value

A list containing 'nodes' (data.frame) and 'edges' (data.frame).

Examples

```
mermaid <- "graph TD\n  A --> B"
parse_mermaid(mermaid)
```

RDSSaver

*RDS File Checkpointer***Description**

A lightweight, file-based checkpointer that uses R's native saveRDS and readRDS functions. Each thread is saved as an individual .rds file in a specified directory.

Value

An RDSSaver object.

Super class

[HydraR::Checkpoint](#) -> RDSSaver

Public fields

dir String. Directory to store .rds checkpoint files.

Methods**Public methods:**

- [RDSSaver\\$new\(\)](#)
- [RDSSaver\\$put\(\)](#)
- [RDSSaver\\$get\(\)](#)
- [RDSSaver\\$clone\(\)](#)

Method new(): Initialize RDSSaver

Usage:

```
RDSSaver$new(dir = "checkpoints")
```

Arguments:

dir String. Directory path for checkpoint files.

Method put(): Persist state to an .rds file.

Usage:

```
RDSSaver$put(thread_id, state)
```

Arguments:

thread_id String.

state AgentState object.

Method `get()`: Load state from an .rds file.

Usage:

`RDSSaver$get(thread_id)`

Arguments:

`thread_id` String.

Returns: AgentState object or NULL.

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

Usage:

`RDSSaver$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# 1. Persistent checkpointing to a local directory
saver <- RDSSaver$new(dir = "vault/checkpoints")

# 2. Create a DAG and run it with a specific thread ID
dag <- dag_create(checkpointer = saver)
dag$run(thread_id = "agent_session_alpha")

# 3. Later, resume the same session - HydraR will load the RDS file
dag$run(thread_id = "agent_session_alpha")

## End(Not run)
```

reducer_append

Built-in Reducer: Append

Description

A functional reducer that appends new elements to an existing vector or list. This is the standard pattern for accumulating results or logs across multiple agent steps.

Usage

`reducer_append(current, new)`

Arguments

<code>current</code>	The current value in the state.
<code>new</code>	The new value to be added.

Value

A combined vector or list containing both current and new.

Examples

```
## Not run:
# Accumulate a trace of agent IDs
state <- AgentState$new(
  initial_data = list(visited = character()),
  reducers = list(visited = reducer_append)
)

state$update(list(visited = "agent_a"))
state$update(list(visited = "agent_b"))

print(state$get("visited")) # [1] "agent_a" "agent_b"

## End(Not run)
```

reducer_merge_list	<i>Built-in Reducer: Merge List</i>
--------------------	-------------------------------------

Description

A functional reducer that performs a deep merge of two named lists. It uses `utils::modifyList` internally, ensuring that existing keys are overwritten by new values while preserving other keys.

Usage

```
reducer_merge_list(current, new)
```

Arguments

current	The current list in the state.
new	The new list to merge in.

Value

A single merged list.

Examples

```
## Not run:
# Deep merging configuration or results
current_cfg <- list(params = list(temp = 0.5, top_p = 1.0), tags = "v1")
new_cfg <- list(params = list(temp = 0.7), tags = "v2")

# modifyList behavior ensured: tags is replaced, params is merged
merged <- reducer_merge_list(current_cfg, new_cfg)
```

```
str(merged)
# list(params = list(temp = 0.7, top_p = 1.0), tags = "v2")

## End(Not run)
```

register_logic	<i>Register Logic Function</i>
----------------	--------------------------------

Description

Stores a pure R function in a centralized logic registry. This allows the function to be referenced by name in Mermaid diagrams or YAML/JSON workflow definitions without needing to pass the function object across environments.

Usage

```
register_logic(name, fn)
```

Arguments

name	String. A unique identifier for the function (e.g., "validate_data").
fn	Function. The R function to be registered. It should typically accept an AgentState or RestrictedState object.

Value

The registry environment (invisibly).

Examples

```
## Not run:
# 1. Register a simple math function
register_logic("add_one", function(state) list(status = "ok", output = state$get("x") + 1))

# 2. Register a complex validation function that uses the Logic Registry
# This function can now be referenced by name 'validate_results' in any YAML workflow.
validate_results <- function(state) {
  results <- state$get("researcher_node")
  if (is.null(results)) {
    return(list(status = "failed", message = "No results found"))
  }
  list(status = "success", output = list(valid = TRUE))
}
register_logic("validate_results", validate_results)

## End(Not run)
```

register_role	<i>Register an LLM Role (System Prompt)</i>
---------------	---

Description

Stores a persistent system prompt or "identity" in the registry. This is useful for centralizing agent personas so they can be reused across multiple DAGs or workflows.

Usage

```
register_role(name, prompt_text)
```

Arguments

name	String. A unique, logic-friendly identifier for the role (e.g., "critic").
prompt_text	String. The full text of the system prompt the LLM should assume.

Value

The registry environment (invisibly).

Examples

```
## Not run:
# Define a specific identity for a bioinformatics analyst
register_role(
  name = "bio_analyst",
  prompt_text = "You are a senior bioinformatician specializing in NGS data.
  Always provide R code for visualization using ggplot2."
)

# Register a reviewer role
register_role(
  name = "reviewer",
  prompt_text = "You are a rigorous peer reviewer. Check for statistical accuracy."
)

## End(Not run)
```

render_workflow_file *Render Workflow Diagram from File*

Description

Loads a workflow from a file and renders its architecture as a Mermaid diagram. Supports high-fidelity exports to various image formats.

Usage

```
render_workflow_file(file_path, output_file = NULL, status = FALSE, ...)
```

Arguments

file_path	String. Path to the YAML workflow file.
output_file	String. Optional path to save the diagram (e.g., "plot.png"). Supported extensions: .png, .pdf, .svg.
status	Logical. If TRUE, styling is applied (requires a valid trace log in the workflow state).
...	Additional arguments passed to 'dag\$run()'.

Value

A 'DiagrammeR' htmlwidget if 'output_file' is NULL, otherwise saves the file.

Examples

```
## Not run:
# 1. View interactive diagram in RStudio
render_workflow_file("plans/main.yaml")

# 2. Export high-resolution diagram for publication
render_workflow_file(
  file_path = "plans/main.yaml",
  output_file = "reports/figures/workflow_v1.png",
  status = TRUE # Colors nodes by their last status
)

## End(Not run)
```

`resolve_default_driver`

<!-- APAF Bioinformatics | factory.R | Approved | 2026-03-30 -> Resolve a Default Driver from Shorthand ID

Description

Provides a mechanism to quickly obtain a pre-configured AgentDriver using logic-friendly keys like "gemini", "claude", or "openai".

Usage

```
resolve_default_driver(driver_id, driver_registry = NULL)
```

Arguments

`driver_id` String. A shorthand identifier. Supported values include: "gemini", "gemini_api", "anthropic", "anthropic_api", "openai", "openai_api", and "ollama".

`driver_registry` DriverRegistry. An optional registry object to look up custom drivers first. If omitted, the global `get_driver_registry()` is used.

Value

An AgentDriver object.

Examples

```
## Not run:
# 1. Simple shorthand resolution
drv1 <- resolve_default_driver("gemini") # GeminiCLIDriver
drv2 <- resolve_default_driver("openai_api") # OpenAIAPIDriver

# 2. Resolution with a custom registry context
reg <- DriverRegistry$new()
reg$register(AnthropicCLIDriver$new(id = "my_custom_claude"))

# This will find the driver in the provided registry
drv3 <- resolve_default_driver("my_custom_claude", driver_registry = reg)

## End(Not run)
```

RestrictedState	<i>Restricted State R6 Class</i>
-----------------	----------------------------------

Description

A security wrapper for AgentState that restricts access based on node_id. Implements "True Privacy" for inter-agent communication.

Value

A 'RestrictedState' object.

Public fields

state AgentState. The underlying global state.
node_id String. The ID of the currently executing node.
read_only Logical. If TRUE, set() and update() are blocked.
logger MessageLog. Audit log for communication.

Methods**Public methods:**

- [RestrictedState\\$new\(\)](#)
- [RestrictedState\\$get\(\)](#)
- [RestrictedState\\$set\(\)](#)
- [RestrictedState\\$update\(\)](#)
- [RestrictedState\\$send_message\(\)](#)
- [RestrictedState\\$receive_messages\(\)](#)
- [RestrictedState\\$clear_inbox\(\)](#)
- [RestrictedState\\$get_all\(\)](#)
- [RestrictedState\\$clone\(\)](#)

Method new(): Initialize RestrictedState

Usage:

RestrictedState\$new(state, node_id, logger = NULL, read_only = FALSE)

Arguments:

state AgentState object.
node_id String node ID.
logger Optional MessageLog object.
read_only Logical. If TRUE, blocks all write operations. Restricted Get

Method get():

Usage:

RestrictedState\$get(key, default = NULL)

Arguments:

key String.

default Default value. Restricted Set

Method set():

Usage:

RestrictedState\$set(key, value)

Arguments:

key String.

value Any. Forward Update

Method update():

Usage:

RestrictedState\$update(updates)

Arguments:

updates List. Send Private Message

Method send_message():

Usage:

RestrictedState\$send_message(to, content, ...)

Arguments:

to String. Target node ID.

content Any. Message content.

... Additional metadata. Receive Private Messages

Method receive_messages():

Usage:

RestrictedState\$receive_messages()

Returns: List of messages for this node. Clear Own Inbox

Method clear_inbox(): Removes all messages from the node's own inbox. Filtered Get All

Usage:

RestrictedState\$clear_inbox()

Method get_all():

Usage:

RestrictedState\$get_all()

Returns: List. All public state variables (no private inboxes).

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

Usage:

RestrictedState\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Secure state wrapper for a specific node
global_state <- AgentState$new(initial_data = list(x = 1))
node_state <- RestrictedState$new(state = global_state, node_id = "agent_01")

# 2. Inter-agent private messaging (True Privacy)
node_state$send_message(to = "agent_02", content = "Secret instructions")

# 3. Recipient reads their own inbox
recipient_state <- RestrictedState$new(state = global_state, node_id = "agent_02")
messages <- recipient_state$receive_messages()

## End(Not run)
```

set_default_driver	<i>Set the Default Agent Driver</i>
--------------------	-------------------------------------

Description

Set the Default Agent Driver

Usage

```
set_default_driver(driver)
```

Arguments

driver AgentDriver object or ID string.

Value

NULL (invisibly)

Examples

```
## Not run:
# 1. Set a global default driver for all LLM nodes
set_default_driver(AnthropicCLIDriver$new(model = "claude-3-opus"))

# 2. Or set it using an ID already present in the registry
get_driver_registry()$register(GeminiCLIDriver$new(id = "fast-gen"))
set_default_driver("fast-gen")

## End(Not run)
```

`spawn_dag`*Spawn an AgentDAG from a Workflow Object*

Description

A high-level helper that orchestrates the "Low Code" lifecycle: it takes a workflow definition (from [load_workflow](#)), parses the internal graph structure, instantiates all nodes via the provided factory, applies conditional/error edges, and performs a final compilation check.

Usage

```
spawn_dag(wf, node_factory = auto_node_factory())
```

Arguments

<code>wf</code>	List. A workflow object previously returned by <code>load_workflow()</code> .
<code>node_factory</code>	Function. An optional factory function to map Mermaid labels to nodes. Defaults to auto_node_factory .

Value

A compiled and ready-to-run AgentDAG object.

Examples

```
## Not run:
# 1. Load the declarative plan
# This registers roles and logic from the YAML into the global registry.
wf <- load_workflow("plans/pipeline.yaml")

# 2. Spawn the executable DAG
# Uses auto_node_factory() to parse the Mermaid string in the YAML.
dag <- spawn_dag(wf)

# 3. Execute with initial data
results <- dag$run(initial_state = wf$initial_state)

## End(Not run)
```

standard_node_factory *Standard Node Factory for Mermaid*

Description

Default mapping of Mermaid labels to AgentNodes. Convention: "type:name" or "name" Types supported: "logic", "llm" (requires global driver).

Usage

```
standard_node_factory(id, label, driver = NULL)
```

Arguments

id	String. Node ID.
label	String. Node label.
driver	AgentDriver. Optional driver for LLM nodes.

Value

AgentNode object.

Examples

```
## Not run:
# 1. Fast resolve from a Mermaid string with defaults
drv <- resolve_default_driver("gemini")
node <- standard_node_factory("n1", "logic:clean_data")

# 2. Using the LLM type with a provided driver
node_llm <- standard_node_factory("n2", "llm:Technical Writer", driver = drv)

## End(Not run)
```

validate_workflow_file

Validate Workflow File Syntax and Consistency

Description

A high-level helper that performs a comprehensive check on a YAML/JSON workflow file. This includes schema validation, topological consistency checks (Mermaid vs YAML), and R logic syntax linting.

Usage

```
validate_workflow_file(file_path)
```

Arguments

file_path String. Path to the workflow definition file.

Value

Logical TRUE if valid (invisibly). Throws a detailed error on failure.

Examples

```
## Not run:
# Holistic check: schema + topology + R linting
validate_workflow_file("inst/workflows/bioinfo.yaml")

## End(Not run)
```

validate_workflow_full

Validate HydraR Workflow Integration

Description

Performs a holistic check on the instantiated DAG and the source workflow. Ensures all roles, logic, and edges are synchronized and syntactically correct.

Usage

```
validate_workflow_full(dag, wf)
```

Arguments

dag AgentDAG object.

wf List. The workflow object from 'load_workflow'.

Value

Logical TRUE if valid, otherwise throws an error.

Examples

```
## Not run:
# Perform a deep audit of an instantiated DAG against its YAML source
wf_data <- load_workflow("orchestration_plan.yaml")
dag <- spawn_dag(wf_data)

# This ensures no 'ghost' edges exist in the Mermaid graph
validate_workflow_full(dag, wf_data)

## End(Not run)
```

WorktreeManager

Git Worktree Manager R6 Class

Description

Manages the lifecycle of isolated git worktrees for parallel DAG branches. This ensures that each parallel worker has its own filesystem checkout, preventing file-level conflicts between concurrent CLI agents.

Value

A ‘WorktreeManager’ R6 object.

Public fields

repo_root String. Path to the main git repository.
worktrees List. Active worktrees indexed by node_id.
base_branch String. The branch to fork worktrees from.
branch_prefix String. Prefix for auto-generated branch names.
cleanup_policy String. "auto", "none", or "aggressive".
thread_id String. Optional identifier for the current DAG execution.

Methods

Public methods:

- [WorktreeManager\\$new\(\)](#)
- [WorktreeManager\\$validate_branch_name\(\)](#)
- [WorktreeManager\\$create\(\)](#)
- [WorktreeManager\\$get_path\(\)](#)
- [WorktreeManager\\$get_branch\(\)](#)
- [WorktreeManager\\$remove_worktree\(\)](#)
- [WorktreeManager\\$delete_branch\(\)](#)
- [WorktreeManager\\$remove\(\)](#)
- [WorktreeManager\\$cleanup_node\(\)](#)
- [WorktreeManager\\$cleanup\(\)](#)
- [WorktreeManager\\$branch_exists\(\)](#)
- [WorktreeManager\\$is_repo_clean\(\)](#)
- [WorktreeManager\\$clone\(\)](#)

Method new(): Initialize the WorktreeManager.

Usage:

```

WorktreeManager$new(
  repo_root = getwd(),
  base_branch = "main",
  branch_prefix = "hydra",
  cleanup_policy = "auto",
  thread_id = NULL
)

```

Arguments:

repo_root Path to the main git repository. Defaults to 'getwd()'.

base_branch Branch to branch from. Defaults to "main".

branch_prefix Prefix for new branches.

cleanup_policy Cleanup strategy.

thread_id Optional string to group worktrees. Defaults to random UUID.

Returns: A new 'WorktreeManager' object. Validate a branch name

Method validate_branch_name():

Usage:

```
WorktreeManager$validate_branch_name(branch)
```

Arguments:

branch String. The branch name to validate.

Returns: Logical or error. Create an Isolated Worktree

Method create():

Usage:

```
WorktreeManager$create(node_id, branch_name = NULL, fail_if_dirty = TRUE)
```

Arguments:

node_id String. Identifier for the DAG node.

branch_name Optional specific branch name.

fail_if_dirty Logical. Whether to stop if repo has uncommitted changes.

Returns: String. The absolute path to the new worktree. Get Worktree Path for a Node

Method get_path():

Usage:

```
WorktreeManager$get_path(node_id)
```

Arguments:

node_id Node identifier.

Returns: String path or NULL. Get Worktree Branch for a Node

Method get_branch():

Usage:

```
WorktreeManager$get_branch(node_id)
```

Arguments:

node_id Node identifier.

Returns: String branch name or NULL. Remove the Physical Worktree Directory

Method remove_worktree():

Usage:

```
WorktreeManager$remove_worktree(node_id, force = TRUE)
```

Arguments:

node_id Node identifier.

force Logical. Force removal.

Returns: Logical (invisibly). Delete the branch and unregister node

Method delete_branch():

Usage:

```
WorktreeManager$delete_branch(node_id)
```

Arguments:

node_id Node identifier.

Returns: Logical (invisibly). Legacy remove method for backward compatibility

Method remove():

Usage:

```
WorktreeManager$remove(node_id, force = TRUE, delete_branch = FALSE)
```

Arguments:

node_id Node identifier.

force Logical. Force removal.

delete_branch Logical. Delete the branch.

Returns: NULL (called for side effect). Cleanup Worktree based on Policy and Status

Method cleanup_node():

Usage:

```
WorktreeManager$cleanup_node(node_id, status = "success")
```

Arguments:

node_id Node identifier.

status String. "success", "failure", or "conflict".

Returns: NULL (called for side effect). Cleanup All Worktrees

Method cleanup(): Removes all worktrees managed by this instance.

Usage:

```
WorktreeManager$cleanup()
```

Returns: Logical (invisibly). Check if a branch exists

Method branch_exists():

Usage:

```
WorktreeManager$branch_exists(branch)
```

Arguments:

branch String.

Returns: Logical. Check if the Repository is Clean

Method is_repo_clean():

Usage:

```
WorktreeManager$is_repo_clean()
```

Returns: Logical.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## Not run:
# 1. Initialize WorktreeManager for a specific repository
wt_manager <- WorktreeManager$new(
  repo_root = "~/Projects/my_repo",
  base_branch = "develop",
  cleanup_policy = "auto"
)

# 2. Create an isolated worktree for a parallel node
wt_path <- wt_manager$create(node_id = "parallel_worker_A")
message("Worktree created at: ", wt_path)

# 3. Cleanup after success
wt_manager$cleanup_node("parallel_worker_A", status = "success")

## End(Not run)
```

Index

add_llm_node, [3](#)
add_logic_node, [5](#)
AgentBashNode, [6](#)
AgentDAG, [7](#)
AgentDriver, [15](#)
AgentLLMNode, [18](#)
AgentLogicNode, [20](#)
AgentMapNode, [21](#)
AgentNode, [23](#)
AgentObserverNode, [25](#)
AgentPythonNode, [26](#)
AgentRouterNode, [28](#)
AgentState, [30](#)
AgentTool, [32](#)
AnthropicAPIDriver, [34](#)
AnthropicCLIDriver, [36](#)
auto_node_factory, [13](#), [38](#), [91](#)

Checkpoint, [39](#)
cleanup_jules_branches, [40](#)
ConflictResolver, [41](#)
CopilotCLIDriver, [43](#)
create_merge_harmonizer, [44](#)

dag_add_llm_node, [45](#)
dag_add_logic_node, [46](#)
dag_create, [47](#)
DriverRegistry, [48](#)
DuckDBMessageLog, [50](#)
DuckDBSaver, [52](#)

extract_r_code_advanced, [53](#)

format_toolset, [54](#)

GeminiAPIDriver, [55](#)
GeminiCLIDriver, [57](#)
GeminiImageDriver, [59](#)
get_agent_roles, [61](#)
get_default_driver, [62](#)
get_driver_registry, [62](#)

get_logic, [63](#)
get_role, [64](#)
get_role_prompt, [64](#)

HydraR::AgentDriver, [34](#), [36](#), [43](#), [55](#), [57](#), [59](#),
[75](#), [77](#), [79](#)
HydraR::AgentNode, [6](#), [18](#), [20](#), [22](#), [25](#), [26](#), [28](#)
HydraR::Checkpoint, [52](#), [71](#), [81](#)
HydraR::GeminiAPIDriver, [59](#)
HydraR::MessageLog, [50](#), [67](#), [70](#)

init_bot_history, [65](#)
is_named_list, [66](#)

JSONLMessageLog, [66](#)

list_logic, [68](#)
load_workflow, [69](#), [91](#)

MemoryMessageLog, [70](#)
MemorySaver, [71](#)
mermaid_to_dag, [72](#)
MessageLog, [73](#)

OllamaDriver, [74](#)
OpenAIAPIDriver, [77](#)
OpenAICodexCLIDriver, [79](#)

parse_mermaid, [80](#)

RDSsaver, [81](#)
reducer_append, [82](#)
reducer_merge_list, [83](#)
register_logic, [84](#)
register_role, [85](#)
render_workflow_file, [86](#)
resolve_default_driver, [87](#)
RestrictedState, [88](#)

set_default_driver, [90](#)
spawn_dag, [91](#)

standard_node_factory, [92](#)

validate_workflow_file, [92](#)

validate_workflow_full, [93](#)

WorktreeManager, [94](#)