Package: admiralagent (via r-universe)

November 28, 2026

Title Spec-Driven ADaM Code Generation with LLM-Translated Layers

Version 0.1.1

Date 2026-10-21

Description Translates Pinnacle 21 (P21)-style ADaM (Analysis Data Model) specifications into executable 'admiral' code with validation comments. An LLM (large language model, optional, via 'ellmer') maps each spec variable's free-text derivation onto a closed vocabulary of derivation layers (a structured JSON intermediate representation (IR)); a deterministic compiler renders the IR into 'admiral'/'metatools' code with 'CHECK' validation comments and writes provenance sidecars. Ships with a zero-dependency rule-based classifier as baseline and fallback; the LLM never writes R code directly and all model output is validated against the closed vocabulary before use. The specification structure follows the CDISC (Clinical Data Interchange Standards Consortium) Analysis Data Model standard <https://www.cdisc.org/standards/foundational/adam>.

License MIT + file LICENSE

Encoding UTF-8

URL https://github.com/yanmingyu92/admiralagent

BugReports https://github.com/yanmingyu92/admiralagent/issues

Depends R (>= 4.1)

Imports glue, jsonlite, digest, stats, utils

Suggests xml2, ellmer, metacore, admiral, metatools, readxl, testthat (>= 3.0.0), rlang, tibble, knitr, rmarkdown, dplyr, haven, pharmaversesdtm, pharmaverseadam

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Jaime Yan [aut, cre]

Maintainer Jaime Yan <yanmingyunmt@gmail.com>

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

Date/Publication 2026-11-28 21:05:07 UTC

RemoteUrl https://github.com/cran/admiralagent

RemoteRef HEAD

RemoteSha a99524834c1cc25e83e138439dff43877f8babfd

Contents

aa_layers . . . 3
accept_model . . . 4
agreement_overstatement . . . 5
artifact_hash . . . 6
build_context . . . 6
build_system_prompt . . . 7
check_admiral_compat . . . 8
classify_variables . . . 8
classify_variables_dual . . . 10
classify_variables_llm . . . 11
codelist_from_data . . . 12
dataset_json_meta . . . 13
dual_disagreements . . . 14
evals_accuracy . . . 14
execute_ir . . . 15
ir_dependency_report . . . 16
layer_docs . . . 17
layer_names . . . 18
load_evals . . . 18
mcp_handle_request . . . 19
mcp_serve . . . 20
mcp_tool_specs . . . 20
mock_metacore . . . 21
mock_spec_adsl . . . 22
new_step . . . 22
new_variable_ir . . . 23
print.aa_agreement . . . 24
print.aa_model_acceptance . . . 24
read_artifact . . . 25
read_dataset_json . . . 26
read_define . . . 27
read_spec . . . 28
read_spec_df . . . 29
render_program . . . 30
render_variable . . . 31
routing_plan . . . 32
run_agreement . . . 33

run_evals 34
run_validation 34
spec_variables 35
validate_ir 36
write_artifact 37
write_program_artifact 38

Index **40**

`aa_layers` *Layer registry*

Description

Returns the single source of truth mapping each derivation layer to its admiral/metatools function, argument schema, required args, render closure, check ids, and ordering rank. The LLM output space is the cartesian product of this registry; nothing outside it passes [validate_ir()].

Usage

```
aa_layers()
```

Details

Layer registry for admiralagent

Value

A named list; one entry per layer with elements ‘fn‘, ‘label‘, ‘args‘, ‘required‘, ‘rank‘, ‘checks‘, ‘inputs‘, ‘render‘, and - where the layer has optional args with an implied value - ‘defaults‘, a named list of literals or functions of the supplied args. Defaults are applied to ‘render‘ input and by canonicalization, so an omitted arg and the same arg stated explicitly are one derivation.

Examples

```
reg <- aa_layers()
length(reg)
reg$assign$fn
```

`accept_model` *Model acceptance gate*

Description

Model+harness acceptance gate. A harness does not transfer across models, so every candidate model must be accepted before it is allowed to translate specs: the candidate runs the spec-to-IR regression corpus ([run_evals()] machinery) and is cross-compared against the deterministic rules baseline at full canonical semantics ([run_agreement()]). Both thresholds are required arguments with no defaults - acceptance criteria must be prespecified by the deploying organization, not inherited from the tool.

The decision is fail-closed: any pipeline error rejects the candidate. Acceptance certifies translation consistency against a self-scoring corpus - not correctness, not fitness for a particular study. Accepted output still passes through ‘validate_ir()‘, the dual-backend consensus, and human review like any other IR.

Usage

```
accept_model(
  chat,
  dataset = "ADSL",
  min_eval_accuracy,
  min_full_agreement,
  prompt_variant = c("full", "neutral", "minimal"),
  model_label = NULL
)
```

Arguments

`chat`	ellmer chat object (or any object with a ‘chat(prompt)‘ method) for the candidate model.
`dataset`	Dataset tag of the corpus cases to run (default ‘"ADSL"‘).
`min_eval_accuracy`	Required minimum fraction of corpus cases whose produced IR matches the expected fingerprint (args-aware).
`min_full_agreement`	Required minimum agreement rate between the candidate and the rules baseline at ‘ir_fingerprint(level = "full")‘.
`prompt_variant`	System prompt variant used for the candidate run.
`model_label`	Optional label for reports and the audit record; when ‘NULL‘, the chat object’s ‘get_model()‘ is tried, else ‘"unknown"‘.

Details

Accept or reject an LLM backend for spec translation

Value

An ‘aa_model_acceptance‘ list: ‘accepted‘ (logical), ‘reasons‘ (failed criteria, empty when accepted), ‘metrics‘ (‘eval_accuracy‘, ‘full_agreement‘, ‘layer_chain_agreement‘, ‘overstatement‘), ‘thresholds‘, plus the underlying ‘evals‘ data.frame and ‘agreement‘ object. Printed via ‘print.aa_model_acceptance()‘.

Examples

```
## Not run:
# Not executable without an LLM provider API key (ellmer chat object):
chat <- ellmer::chat_deepseek(model = "deepseek-chat")
accept_model(chat, min_eval_accuracy = 0.8, min_full_agreement = 0.5)

## End(Not run)
```

---

agreement_overstatement

How much an args-blind label overstates agreement

Description

The defect this harness exists to quantify: comparing only the layer-name sequence counts two voters as agreeing whenever they pick the same layers, even when every argument differs. This reports the gap between the ‘layer_chain‘ and ‘full‘ agreement rates - i.e. the share of cases the old [evals_layers_label()] comparison would have scored as agreement but which differ semantically.

The result inherits every caveat of [run_agreement()]: it is computed pre-execution over voters reading the same spec text from a self-scoring regression corpus, so it quantifies overstated translation consistency, not correctness and not independent double programming.

Usage

```
agreement_overstatement(res)
```

Arguments

`res`	An ‘aa_agreement‘ object from ‘run_agreement()‘.

Value

A list with ‘layer_chain‘, ‘full‘, ‘overstatement‘ (the difference) and ‘cases‘ (the ids counted as agreeing only at ‘layer_chain‘).

Examples

```
res <- run_agreement(list(a = "rules", b = "rules"), dataset = "ADSL")
agreement_overstatement(res)
```

artifact_hash *Artifact hash*

Description

Hashes the canonical JSON of the IR (preserving variable and step order); used in deterministic artifact file names so identical IRs always overwrite the same files. The package version is deliberately NOT part of the hash: it is recorded as provenance in sidecars/manifests instead, so a version bump never silently moves artifact identities.

Usage

```
artifact_hash(ir)
```

Arguments

ir	List of ‘aa_variable_ir‘ objects.

Details

Deterministic hash of a variable IR

Value

An 8-character hash string.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
artifact_hash(ir)
```

build_context *Build LLM context*

Description

Converts spec rows into a per-variable list of schema fields (dataset, variable, label, type, origin, derivation, source pointers). No patient data ever enters the context. The dataset field lets the model emit the correct target-dataset token for non-ADSL specs; without it the model can only guess (and copies the system prompt's ADSL example), so [align_batch()] rejected those batches (finding F-06).

Usage

```
build_context(vars)
```

Arguments

vars	Spec data frame subset from [spec_variables()].

Details

Build schema-only LLM context

Value

A named list (by variable name) of schema-only field lists.

Examples

```
ctx <- build_context(spec_variables(mock_spec_adsl(), "ADSL"))
ctx$STUDYID$derivation
```

---

build_system_prompt *Build system prompt*

Description

Assembles the translation prompt: role definition, the full layer vocabulary from [layer_docs()], the strict JSON output contract with an example object, and the hard rules (closed vocabulary, ‘on‘ placement, formula whitelists, merge_var for cross-dataset values, needs_human abstention).

Usage

```
build_system_prompt()
```

Details

Build the LLM system prompt

Value

A single character string.

Examples

```
prompt <- build_system_prompt()
nchar(prompt)
```

check_admiral_compat *admiral compatibility check*

Description

Compares the minimum admiral version required by the layers used in the IR against the installed admiral version. Absence of admiral is not an error: rendering is static and only execution is blocked.

Usage

```
check_admiral_compat(ir, installed = NA)
```

Arguments

ir	List of ‘aa_variable_ir‘ objects.
installed	admiral version to compare against; defaults to the installed ‘utils::packageVersion("admiral")‘ (NA-safe when missing).

Details

Check admiral version compatibility of an IR

Value

A character vector of problem messages (empty when compatible); the vector carries a ‘note‘ attribute when admiral is not installed.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
check_admiral_compat(ir)
```

classify_variables *Classify spec variables*

Description

Translates each spec variable’s free-text derivation into a variable IR using the selected backend: ‘"rules"‘ runs the built-in deterministic, zero-dependency keyword classifier (baseline and fallback); ‘"llm"‘ runs the structured-extraction pipeline via [classify_variables_llm()] (requires an ellmer ‘chat‘).

Usage

```
classify_variables(
  spec,
  dataset,
  backend = c("rules", "llm"),
  chat = NULL,
  samples = 1,
  consensus = c("majority", "first"),
  prompt_variant = c("full", "neutral", "minimal")
)
```

Arguments

spec	Spec data frame compatible with [read_spec_df()].
dataset	Dataset name to classify (e.g. ‘"ADSL"‘).
backend	One of ‘"rules"‘ or ‘"llm"‘.
chat	ellmer chat object; required when ‘backend = "llm"‘, ignored otherwise.
samples	Number of independent LLM pipeline runs; used when ‘backend = "llm"‘, ignored otherwise. See [classify_variables_llm()].
consensus	Voting strategy across samples; used when ‘backend = "llm"‘, ignored otherwise. See [classify_variables_llm()].
prompt_variant	System prompt variant; used when ‘backend = "llm"‘, ignored otherwise. See [classify_variables_llm()].

Details

Classify spec variables into layer IR

Value

List of ‘aa_variable_ir‘ objects, one per spec variable.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
ir[[1]]$variable
```

`classify_variables_dual` *Dual-backend consensus classification*

Description

Runs the deterministic rules classifier and the LLM pipeline independently on the same spec, then compares the two translations per variable at ‘ir_fingerprint(level = "full")‘ - complete canonical semantics, with registry defaults filled and intermediate names alpha-renamed, so a rule’s ‘EXST_TRTSDTM‘ temp and an LLM’s ‘EXSTDTM‘ compare equal when the derivation means the same thing.

Agreement adopts the rules IR (deterministic precedence; the semantics are identical anyway). Any divergence - different canonical semantics, or exactly one side abstaining - escalates the variable to ‘needs_human‘ with a rationale naming both layer chains. No side is ever silently preferred: a dual run never derives a variable that a rules-only run would have refused, and never accepts an LLM answer the rules disagreed with. Agreement is corroboration, not proof of correctness.

Usage

```
classify_variables_dual(
  spec,
  dataset,
  chat,
  samples = 1,
  prompt_variant = c("full", "neutral", "minimal")
)
```

Arguments

`spec`	Spec data frame compatible with [read_spec_df()].
`dataset`	Dataset name to classify (e.g. ‘"ADSL"‘).
`chat`	ellmer chat object (or any object with a ‘chat(prompt)‘ method) for the LLM side.
`samples`	Number of independent LLM pipeline runs, passed to [classify_variables_llm()]; ‘> 1‘ gives the LLM side its own internal majority vote before the cross-backend comparison.
`prompt_variant`	System prompt variant for the LLM side; see [classify_variables_llm()].

Details

Classify spec variables with two independent backends

Value

List of ‘aa_variable_ir‘ objects, one per spec variable, with two attributes: ‘dual‘, a data.frame with columns ‘variable‘, ‘class‘ (‘agree‘, ‘disagree‘, ‘rules_abstain‘, ‘llm_abstain‘), ‘rules_fp‘, ‘llm_fp‘ (full fingerprints), ‘rules_layers‘, ‘llm_layers‘ (layer chains), and ‘llm_suggestion‘ (whether the LLM side produced a candidate the reviewer can inspect); ‘agreement_rate‘, the fraction of variables where both backends produced the same canonical semantics; and ‘llm_candidates‘, a named list holding the LLM-side IR for every escalated variable that has one (the reviewer's starting point; never rendered or executed without a human decision).

Examples

```
## Not run:
# Not executable without an LLM provider API key (ellmer chat object):
chat <- ellmer::chat_deepseek(model = "deepseek-chat")
ir <- classify_variables_dual(mock_spec_adsl(), "ADSL", chat)
dual_disagreements(ir) # review queue: every variable the backends split on

## End(Not run)
```

---

classify_variables_llm

Classify spec variables via an LLM chat backend

Description

Runs the batch translate-validate-retry pipeline. With ‘samples > 1‘ the pipeline is repeated and per-variable signatures (step-layer chains) are voted on to counter LLM output variance.

Usage

```
classify_variables_llm(
  vars,
  chat,
  max_attempts = 2,
  batch_size = 4,
  samples = 1,
  consensus = c("majority", "first"),
  prompt_variant = c("full", "neutral", "minimal")
)
```

Arguments

vars	spec data frame from [read_spec_df()].
chat	ellmer chat object (or any object with a ‘chat(prompt)‘ method).
max_attempts	validation-repair attempts per batch.
batch_size	variables per LLM batch.

samples	number of independent pipeline runs; ‘1‘ (default) keeps the classic single-run behavior; ‘> 1‘ enables consensus mode.
consensus	‘majority‘ = per-variable majority vote across samples; ties or all-failed samples mark the variable ‘needs_human‘. ‘first‘ = skip voting and take the earliest successful sample.
prompt_variant	system-prompt variant: ‘full‘ (default) carries the complete convention block; ‘neutral‘ and ‘minimal‘ strip it down for agreement measurement, so consensus reflects independent translations rather than prompt compliance.

Value

List of ‘aa_variable_ir‘ objects. For ‘samples > 1‘ the list carries a ‘consensus‘ attribute: a data.frame with columns ‘variable‘, ‘signature_votes‘ (e.g. ‘"impute_dtc->merge_var:2"‘), ‘chosen‘, ‘unanimous‘.

Examples

```
## Not run:
# Requires the 'ellmer' package and a provider API key:
chat <- ellmer::chat_anthropic()
vars <- spec_variables(mock_spec_adsl(), "ADSL")
ir <- classify_variables_llm(vars, chat)

## End(Not run)
```

---

codelist_from_data *Build a codelist data.frame from observed data values*

Description

Builds a minimal ‘code‘/‘decode‘ codelist data.frame from the unique values of ‘data[[var]]‘: ‘decode‘ is ‘sort(unique(...))‘ of the non-missing values and ‘code‘ is ‘seq_along(decode)‘. The result is suitable as an entry of the ‘codelists‘ argument of [mock_metacore()], e.g. ‘codelists = list(RACE = codelist_from_data("RACE", dm))‘.

Usage

```
codelist_from_data(var, data)
```

Arguments

var	Character, name of the column in ‘data‘ holding the decode values (e.g. ‘"RACE"‘).
data	A data.frame containing ‘var‘.

Value

A data.frame with columns ‘code‘ (integer) and ‘decode‘ (character), one row per unique non-missing value, sorted by decode.

Examples

```
dm <- data.frame(RACE = c("WHITE", "ASIAN", "BLACK", "ASIAN"))
codelist_from_data("RACE", dm)
```

`dataset_json_meta` *Dataset-JSON header metadata*

Description

Reads only the header of a CDISC Dataset-JSON file (version 1.1 or the nested 1.0 layout) and returns the metadata needed to inspect a delivery without materialising the rows: schema version, record count, dataset name/label, provenance fields (‘fileOID‘, ‘originator‘, ‘sourceSystem‘, ‘studyOID‘), and the ‘columns‘ table (itemOID, name, label, dataType, targetDataType, length).

Usage

```
dataset_json_meta(path)
```

Arguments

`path`	Path to a Dataset-JSON file.

Details

Dataset-JSON header metadata

Value

A list with elements ‘path‘, ‘datasetJSONVersion‘, ‘records‘, ‘name‘, ‘label‘, ‘itemGroupOID‘, ‘fileOID‘, ‘originator‘, ‘sourceSystem‘, ‘studyOID‘, and ‘columns‘ (data.frame). Stops with an informative error when required keys are absent.

Examples

```
doc <- list(
  datasetJSONVersion = "1.1",
  datasetJSONCreationDateTime = "2026-01-01T00:00:00",
  columns = list(list(name = "STUDYID", dataType = "string", length = 6)),
  rows = list(list("STUDY01"))
)
f <- file.path(tempdir(), "dm.json")
writeLines(jsonlite::toJSON(doc, auto_unbox = TRUE, null = "null"), f)
meta <- dataset_json_meta(f)
meta$datasetJSONVersion
meta$columns
```

dual_disagreements *Dual-backend disagreements*

Description

Returns the rows of the ‘dual‘ attribute where the two backends did not agree - the variables a human must decide on. Each row carries both layer chains and whether the LLM side produced a candidate IR the reviewer can inspect as a starting point.

Usage

```
dual_disagreements(ir)
```

Arguments

ir	IR list returned by [classify_variables_dual()].

Details

Extract the dual-backend review queue

Value

A data.frame with columns ‘variable‘, ‘class‘, ‘rules_layers‘, ‘llm_layers‘, ‘llm_suggestion‘; zero rows when every variable agreed.

Examples

```
## Not run:
# Needs an ellmer chat object (LLM provider API key); see
# classify_variables_dual():
ir <- classify_variables_dual(mock_spec_adsl(), "ADSL", chat)
dual_disagreements(ir)

## End(Not run)
```

evals_accuracy *Accuracy of an eval run*

Description

Extracts the accuracy from a result data.frame returned by [run_evals()].

Usage

```
evals_accuracy(res)
```

Arguments

res	data.frame with a logical ‘ok‘ column, as returned by [run_evals()].

Value

A single numeric in ‘[0, 1]‘: the share of cases classified as expected.

Examples

```
res <- run_evals(backend = "rules", dataset = "ADSL")
evals_accuracy(res)
```

---

execute_ir *Execute a Layer IR against source datasets*

Description

Runs each variable's rendered code block ([render_variable()]) inside a fresh environment with baseenv() as parent and reports a per-variable status. This is the programmatic replacement for the demo glue loops: idempotent, re-runnable, and safe to call on an IR that mixes executable variables with ‘needs_human‘ review variables.

Contract: - ‘sources‘ must contain the base for the target dataset under the name ‘base‘ (preferred) or under the target dataset name lower-cased (e.g. ‘adsl‘ for target ‘ADSL‘). It is the one-row-per-subject starting point and is assigned into the execution environment as the target dataset object. - Every other element of ‘sources‘ is assigned into the environment under its list name, e.g. ‘list(base = dm, ex = ex, vs = vs_bds)‘ creates ‘ADSL <- base‘, ‘ex‘, ‘vs_bds‘. Step args like ‘on = "ex"‘ or ‘dataset_add = "ex"‘ must match those names. A ‘codelist_var‘ step needs a metacore object named ‘mc‘; pass it as ‘sources$mc‘ (typically a [metacore::select_dataset()] subset of [mock_metacore()] output). - Each variable executes transactionally: errors roll back all its changes to target and source datasets, and dependent variables report ERROR. - Before each step's code runs, an idempotency guard removes columns (and computed PARAMCD rows) that this variable's steps would create, from the objects those steps land on - including imputation temp columns and their flag columns on foreign datasets. Calling ‘execute_ir()‘ twice therefore yields identical output, and a partially failed run can be retried without "already exists" warnings or stale values.

The ‘admiralagent‘ layer templates render bare ‘exprs()‘/‘params()‘ helpers; if ‘rlang‘/‘admiral‘ are installed these are pre-assigned into the environment, otherwise the affected variables report ‘ERROR‘.

Usage

```
execute_ir(
  ir,
  sources = list(),
  variables = NULL,
  quiet = TRUE,
  log_file = aa_log_file()
)
```

Arguments

ir	Non-empty list of ‘aa_variable_ir‘ objects (as produced by [classify_variables()]).
sources	Named list of source data objects; must contain ‘base‘ (or the lower-cased target dataset name). See the Description for the full contract.
variables	Optional character vector restricting execution to these spec variables (default all). Only selected variables run; required prior results must be supplied in sources. Unknown names are rejected.
quiet	If ‘TRUE‘ (default), warnings from the evaluated blocks are suppressed and nothing is printed; if ‘FALSE‘, warnings pass through and the status table is printed.
log_file	Audit log destination. Logging is ON by default and writes one hash-chained record per variable (see [log_run()]) plus exactly one ‘run_manifest‘ record for the call, carrying that call’s ‘run_id‘ so the manifest joins the executions it describes; the default destination comes from ‘getOption("admiralagent.log_file")‘, then ‘ADMIRALAGENT_LOG_FILE‘, then a session file in [tempdir()]. Pass ‘NA‘ to switch logging off for this call - that suppresses the manifest too.

Value

A list with elements: - ‘adsl‘: the target dataset after execution (named after the IR’s first dataset, so ‘ADSL‘ for an ADSL IR); - ‘status‘: data.frame with columns ‘variable‘, ‘status‘ (one of ‘EXECUTED‘, ‘ERROR‘, ‘REVIEW‘), ‘note‘ (error message when ‘ERROR‘, otherwise empty or the review reason); - ‘env‘: the execution environment (useful for inspecting mutated source objects such as imputation temps on ‘ex‘).

Examples

```
ir <- list(new_variable_ir("ADSL", "ITTFL",
  steps = list(new_step("assign_conditional", list(
    target = "ITTFL", condition = "AGE >= 18", true_value = "Y"
  )))
))
res <- execute_ir(ir, sources = list(base = data.frame(AGE = c(15, 30))))
res$status
```

---

ir_dependency_report *IR dependency report*

Description

Audits every step’s consumed references (duration endpoints, assign sources, compute_param/summary_record PARAMCD/AVAL expectations) against columns produced by earlier executable steps or whitelisted via ‘known_columns‘, and lists everything unresolved.

Usage

```
ir_dependency_report(ir, known_columns = character())
```

Arguments

ir	List of ‘aa_variable_ir‘ objects.
known_columns	Character vector of column names assumed to pre-exist in the source data.

Details

Report undefined step inputs in an IR

Value

A data.frame with columns ‘variable‘, ‘input‘, ‘dataset‘, ‘issue‘ (zero rows when every input resolves).

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
ir_dependency_report(ir, known_columns = c("STUDYID", "USUBJID", "SUBJID",
  "TRT01P", "AGE", "RACEN", "RFSTDTC", "RFENDTC", "EXSTDTC"))
```

---

layer_docs *Layer documentation cards*

Description

Renders each registry entry (function, label, args, required args) into a compact text card. These cards are embedded verbatim in the LLM system prompt built by [build_system_prompt()].

Usage

```
layer_docs()
```

Details

Human-readable documentation of every layer

Value

Named character vector, one documentation block per layer.

Examples

```
cards <- layer_docs()
cat(cards[["assign"]])
```

`layer_names` *Layer names*

Description

Convenience accessor over [aa_layers()] returning the allowed ‘layer‘ vocabulary.

Usage

```
layer_names()
```

Details

Names of all registered derivation layers

Value

Character vector of layer names.

Examples

```
layer_names()
```

`load_evals` *Load the spec-to-IR classification eval corpus*

Description

Reads ‘inst/evals/spec-to-ir.jsonl‘, one JSON eval case per line, into a named list. Works both for an installed package (via [system.file()]) and from the package source tree (walking up from the working directory to find ‘inst/evals/‘), so tests run uninstalled.

Usage

```
load_evals(path = NULL)
```

Arguments

`path`	Optional full path to a ‘.jsonl‘ corpus file. Defaults to the bundled corpus located automatically.

Value

A named list of parsed eval cases (names are case ids). Each case has fields ‘id‘, ‘dataset‘, ‘spec_row‘, ‘expected_layers‘, ‘expected_needs_human‘, and ‘note‘.

Examples

```
cases <- load_evals()
length(cases)
cases[[1]]$id
```

`mcp_handle_request` *Handle one MCP / JSON-RPC 2.0 request*

Description

Dispatches a parsed JSON-RPC 2.0 request list against the admiralagent MCP methods: ‘initialize‘, ‘notifications/initialized‘, ‘tools/list‘, and ‘tools/call‘. Tool results are wrapped as MCP text content ({content: [{type: "text", text: <json>}], isError: false}); tool execution errors come back with ‘isError: true‘. Unknown methods return a JSON-RPC error (-32601), structurally invalid requests -32600. The function is pure: it never touches stdin/stdout, so the whole server can be unit-tested in-process.

Usage

```
mcp_handle_request(req)
```

Arguments

`req`	Parsed request: a JSON-RPC 2.0 object as an R list (typically ‘jsonlite::fromJSON(line, simplifyVector = FALSE)‘).

Value

A JSON-RPC response list, or ‘NULL‘ for notifications (no response may be sent).

Examples

```
resp <- mcp_handle_request(list(
  jsonrpc = "2.0", id = 1, method = "tools/list"
))
vapply(resp$result$tools, function(t) t$name, character(1))
```

mcp_serve *Run the admiralagent MCP stdio server*

Description

Blocking newline-delimited JSON-RPC 2.0 loop over stdin/stdout (MCP stdio transport, protocol version 2024-11-05). One request per line in, zero or one response line out, flushed immediately. EOF on stdin ends the loop cleanly. Refuses to run in interactive sessions so an accidental call cannot hang the console. Meant to be launched via 'Rscript tools/mcp_server.R'.

Usage

```
mcp_serve()
```

Value

Invisible 'NULL' when stdin reaches EOF.

Examples

```
## Not run:
# Blocking stdio loop; cannot run inside R CMD check or an
# interactive session. Test single requests in-process with
# mcp_handle_request() instead:
mcp_serve()

## End(Not run)
```

mcp_tool_specs *Tool specifications for the admiralagent MCP server*

Description

Defines the five deterministic tools exposed over MCP stdio: 'aa_classify', 'aa_validate_ir', 'aa_render_program', 'aa_dependency_report', and 'aa_evals'. Every tool is pure base-package logic (no network, no LLM): spec rows in, layer IR / admiral code / validation problems out.

Usage

```
mcp_tool_specs()
```

Value

A named list of tool definitions, each with 'name', 'description', and 'inputSchema' (a JSON-Schema-style list).

Examples

```
tools <- mcp_tool_specs()
names(tools)
```

`mock_metacore` *Build a minimal executable metacore object from a spec data frame*

Description

Constructs a ‘Metacore‘ object via [metacore::metacore()] from a ‘read_spec_df()‘-compatible spec data frame plus user-supplied codelists, so that ‘codelist_var‘ layer code (i.e. [metatools::create_var_from_codelist()]) runs in demos/tests without a vendor P21 spec.

Codelist linking: a spec variable is linked to codelist ‘N‘ when the spec ‘codelist‘ column equals ‘N‘, or when the variable’s derivation text mentions ‘N‘ (word boundary, case-insensitive; e.g. "using the RACE codelist" links to ‘RACE‘). Real decodes cannot be invented from the spec text: supply ‘codelists‘ (built by hand or with [codelist_from_data()]); variables whose derivation mentions a codelist but have no matching entry get no ‘code_id‘ and emit a warning.

When the spec covers exactly one dataset, the returned object is already subsetted with [metacore::select_dataset()] so it can be passed directly as ‘metacore = mc‘ to metatools; with multiple datasets the full object is returned and the caller subsets it.

Usage

```
mock_metacore(spec_df, codelists = list())
```

Arguments

`spec_df`	Spec data.frame compatible with [read_spec_df()] (columns dataset, variable, label, type, origin, derivation; optional length, codelist).
`codelists`	Named list of codelists, each a data.frame with columns ‘code‘ and ‘decode‘ (e.g. ‘list(RACE = codelist_from_data("RACE", dm))‘). The list name becomes the metacore ‘code_id‘.

Value

A ‘Metacore‘ object (single-dataset specs: subsetted via ‘select_dataset()‘; multi-dataset specs: full object). NULL with a warning if ‘metacore‘ is not installed.

Examples

```
## Not run:
# Requires the 'metacore' package:
mc <- mock_metacore(mock_spec_adsl(),
  codelists = list(RACE = data.frame(code = 1:3, decode = c("ASIAN", "BLACK", "WHITE")))
)

## End(Not run)
```

mock_spec_adsl *Mock ADSL spec*

Description

A 10-variable mock ADSL spec (STUDYID, USUBJID, SUBJID, TRT01P, TRTSDTM, TRTEDTM, AGE, AGEGR1, RACEN, BMIBL) with realistic free-text derivations, used in demos, tests, and the vignette.

Usage

```
mock_spec_adsl()
```

Details

Mock ADSL specification

Value

A spec data frame as produced by [read_spec_df()].

Examples

```
spec <- mock_spec_adsl()
spec$variable
```

new_step *New layer step*

Description

Builds one step of a variable's derivation pipeline: a layer name plus its named args. Nested one-element lists are flattened (JSON round-trip normalization) and empty args are dropped.

Usage

```
new_step(layer, args = list())
```

Arguments

layer	Layer name; must be one of [layer_names()].
args	Named list of layer arguments.

Details

Construct a layer step

Value

A list of class ‘aa_step‘ with elements ‘layer‘ and ‘args‘.

Examples

```
new_step("assign", list(target = "TRT01P", from = "TRT01P"))
```

`new_variable_ir` *New variable IR*

Description

Builds the intermediate representation for one spec variable: dataset, variable name, ordered [new_step()] pipeline, plus provenance fields (verbatim spec text, confidence, review flag, rationale).

Usage

```
new_variable_ir(
  dataset,
  variable,
  steps = list(),
  spec_origin = NA_character_,
  confidence = 1,
  needs_human = FALSE,
  rationale = NA_character_
)
```

Arguments

`dataset`	Dataset the variable belongs to (e.g. ‘"ADSL"‘).
`variable`	Variable name (e.g. ‘"TRTSDTM"‘).
`steps`	List of ‘aa_step‘ objects in execution order.
`spec_origin`	Verbatim derivation text from the spec.
`confidence`	Classifier confidence in ‘[0, 1]‘.
`needs_human`	Logical; ‘TRUE‘ routes the variable to human review instead of code generation.
`rationale`	Free-text explanation of the chosen layer chain.

Details

Construct a variable IR

Value

A list of class ‘aa_variable_ir‘.

Examples

```
v <- new_variable_ir(
  dataset = "ADSL",
  variable = "TRTSDTM",
  steps = list(new_step("assign", list(target = "TRTSDTM", literal = "2026-01-01")))
)
```

`print.aa_agreement` *Print an agreement report*

Description

Prints the dataset, voters, and per-level agreement rates of a [run_agreement()] result, followed by a one-line reminder that pre-execution agreement over a self-scoring regression corpus measures translation consistency, not correctness and not independent double programming.

Usage

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

Arguments

`x`	An ‘aa_agreement‘ object from [run_agreement()].
`...`	Ignored.

Value

‘x‘, invisibly.

Examples

```
res <- run_agreement(list(a = "rules", b = "rules"), dataset = "ADSL")
print(res)
```

`print.aa_model_acceptance`
Print a model acceptance report

Description

Print a model acceptance report

Usage

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

Arguments

x	An ‘aa_model_acceptance‘ object from [accept_model()].
...	Ignored.

Value

‘x‘, invisibly.

`read_artifact` *Read artifact*

Description

Parses a sidecar JSON, renormalizes its IR through [new_step()]/[new_variable_ir()] (including step-level ‘on‘ absorption), and returns the IR with its provenance fields. Stale or malformed sidecars surface as errors rather than silently loading.

Reloading also re-checks integrity, in this order: the sidecar still names the artifact it was read as; the IR it carries still hashes to the value the artifact was sealed and named under; the rendered ‘.R‘ beside it still digests to the value the sidecar sealed; and any attached approval gate is well formed and unedited. Each check is a hard error. Together they detect drift between sidecar, code and file name - they are not signatures, and the identity anchor for an approval remains the gate record in the hash-chained audit log.

Usage

```
read_artifact(path)
```

Arguments

path	Path to the sidecar ‘.json‘ file.

Details

Read an artifact sidecar back into an IR

Value

A list with elements ‘ir‘, ‘artifact‘, ‘backend‘, ‘model‘, ‘created_at‘, ‘validation‘, ‘gate‘ (‘NULL‘ when ungated), ‘release_grade‘ (‘"gated"‘ or ‘"ungated-draft"‘), ‘release_ready‘ and ‘gate_enforced‘ (the policy in force when the artifact was written, ‘NA‘ for artifacts written before that was recorded).

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
out <- file.path(tempdir(), "aa-examples-read")
write_artifact(ir, dir = out)
art <- read_artifact(file.path(out, list.files(out, pattern = "[.]json$")[1]))
art$backend
```

`read_dataset_json` *Read a Dataset-JSON file*

Description

Parses a file conforming to the CDISC Dataset-JSON open standard (Data Exchange, version 1.1) and returns the data as a data frame, so Dataset-JSON deliveries can be dropped straight into [execute_ir()] ‘sources‘. Both layouts are supported: the 1.1 single-dataset layout (top-level ‘columns‘/‘rows‘, as written by the R Consortium submission pilots) and the older nested ‘itemGroupData‘ layout (the first item group is read).

Type coercion follows the declared ‘dataType‘ of each column:

- ‘"string"‘/‘"character"‘/‘"URI"‘: character - ‘"integer"‘/‘"float"‘/‘"double"‘/‘"decimal"‘: numeric (double; integers are deliberately not narrowed to R integers to avoid 32-bit overflow on large keys) - ‘"boolean"‘: logical - ‘"date"‘/‘"datetime"‘/‘"time"‘: character by default. If the column declares ‘targetDataType‘ ‘"Date"‘ or ‘"Datetime"‘, a typed conversion via [as.Date()] / [as.POSIXct()] (UTC) is attempted; when any non-missing value fails to parse (e.g. partial SDTM ‘–DTC‘ dates such as ‘"2010-04"‘), the whole column safely stays character. Whenever a typed conversion succeeds, the original JSON strings are preserved in ‘attr(column, "raw")‘.

JSON ‘null‘ cell values become ‘NA‘; numeric columns also accept the legacy pilot string sentinel ‘"NA"‘ (with surrounding whitespace). Other invalid numeric values and non-scalar cells are rejected. A file with ‘records = 0‘ may omit ‘rows‘ entirely; the result is then a zero-row data frame with the declared columns.

Attached metadata: ‘attr(df, "dataset_label")‘ (the item group / dataset label, ‘NA‘ when absent), ‘attr(df, "labels")‘ (named list of variable labels in column order), and ‘attr(df, "dataset_json_version")‘.

Usage

```
read_dataset_json(path)
```

Arguments

`path`	Path to a Dataset-JSON file (‘.json‘).

Details

Read a CDISC Dataset-JSON file

Value

A data.frame with columns in the declared order, carrying the metadata attributes described above. Stops with an informative error when the file is missing, unparsable, or lacks required Dataset-JSON schema keys (‘datasetJSONVersion‘, ‘columns‘, ‘rows‘ when ‘records > 0‘).

Examples

```
# Write a minimal one-column Dataset-JSON file to a temp directory:
doc <- list(
  datasetJSONVersion = "1.1",
  datasetJSONCreationDateTime = "2026-01-01T00:00:00",
  columns = list(list(name = "STUDYID", dataType = "string", length = 6)),
  rows = list(list("STUDY01"))
)
f <- file.path(tempdir(), "dm.json")
writeLines(jsonlite::toJSON(doc, auto_unbox = TRUE, null = "null"), f)
dm <- read_dataset_json(f)
dm$STUDYID
```

`read_define` *Read define.xml*

Description

Primary path: parses the define.xml directly with [parse_define()] (requires the ‘xml2‘ package), which preserves the per-variable derivation text from ‘MethodDef‘ elements - metadata that [metacore::define_to_metacore()] drops for some define versions (e.g. the CDISC pilot submission defines). Fallback path: when the direct parser fails (missing ‘xml2‘, malformed XML, non-ODM structure), the file is re-parsed via metacore and flattened through ‘spec_as_df()‘, whose ‘derivation‘ column may then be empty. When both paths fail an informative error is raised.

Usage

```
read_define(path)
```

Arguments

`path`	Path to the define.xml file.

Details

Read a define.xml metadata file

Value

A spec data frame as produced by [read_spec_df()]; the primary path additionally returns ‘codelist_oid‘ and ‘order‘ columns.

Examples

```
# Write a minimal one-variable define.xml to a temp directory and read it:
define_txt <- paste(
  '<?xml version="1.0" encoding="UTF-8"?>',
  '<ODM ODMVersion="1.3.2" xmlns="http://www.cdisc.org/ns/odm/v1.3">',
  '  <Study OID="STD1"><MetaDataVersion OID="MDV1">',
  '    <ItemGroupDef OID="IG.ADSL" Name="ADDSL" SASDatasetName="ADSL">',
  '      <Description><TranslatedText>Subject-Level</TranslatedText></Description>',
  '      <ItemRef ItemOID="IT.STUDYID" OrderNumber="1" MethodOID="MT.STUDYID"/>',
  '    </ItemGroupDef>',
  '    <ItemDef OID="IT.STUDYID" Name="STUDYID" DataType="text" Length="12">',
  '      <Description><TranslatedText>Study Identifier</TranslatedText></Description>',
  '    </ItemDef>',
  '    <MethodDef OID="MT.STUDYID" Name="STUDYID" Type="Computation">',
  '      <Description><TranslatedText>',
  '        Copied directly from DM.STUDYID',
  '      </TranslatedText></Description>',
  '    </MethodDef>',
  '  </MetaDataVersion></Study>',
  '</ODM>',
  sep = "\n"
)
f <- file.path(tempdir(), "define.xml")
writeLines(define_txt, f)
spec <- read_define(f)
spec[, c("dataset", "variable", "derivation")]
```

---

read_spec *Read spec workbook*

Description

Parses a specifications.xlsx-style workbook via [metacore::spec_to_metacore()] and flattens the result into the package's spec data frame format. Requires the 'metacore' and 'readxl' packages.

Usage

```
read_spec(
  path,
  dataset = NULL,
  where_sep_sheet = "Datasets",
  verbose = "silent"
)
```

Arguments

path	Path to the specification workbook.
dataset	Optional dataset name to subset to.

where_sep_sheet	Whether the where clauses are on a separate sheet.
verbose	metacore verbosity, one of ‘"silent"‘, ‘"verbose"‘.

Details

Read a P21-style specification workbook

Value

A spec data frame as produced by [read_spec_df()].

Examples

```
## Not run:
# Requires a specifications.xlsx-style workbook plus the 'metacore' and
# 'readxl' packages:
spec <- read_spec("specifications.xlsx", dataset = "ADSL")

## End(Not run)
```

---

read_spec_df *Normalize spec data frame*

Description

Validates that a flat (already parsed) spec data frame carries the required columns, fills optional columns (‘length‘, ‘codelist‘, ‘source_dataset‘, ‘source_variable‘, ‘order‘) with ‘NA‘ when absent, and defactors all columns. This is the canonical in-memory spec format all downstream functions accept.

Usage

```
read_spec_df(df)
```

Arguments

df	data.frame with columns dataset, variable, label, type, origin, derivation.

Details

Normalize a flat spec data frame

Value

The normalized data frame (error when required columns are missing).

Examples

```
df <- data.frame(
  dataset = "ADSL", variable = "STUDYID", label = "Study Identifier",
  type = "text", origin = "CRF",
  derivation = "Copied directly from DM.STUDYID",
  stringsAsFactors = FALSE
)
read_spec_df(df)
```

`render_program` *Render program*

Description

Assembles the complete dataset program: DISCLAIMER header, library calls, source-dataset placeholders, all variables in dependency-driven order via [order_variables()], and an optional finalize block (metatools/xportr spec-compliance tail). Fails when [check_admiral_compat()] reports a version conflict.

Usage

```
render_program(
  ir,
  backend_label = "rules",
  finalize = TRUE,
  language = "r",
  gate = NULL
)
```

Arguments

`ir`	Non-empty list of ‘aa_variable_ir‘ objects.
`backend_label`	Backend label recorded in the header (‘"rules"‘ or ‘"llm"‘).
`finalize`	Whether to append the commented finalize/export block.
`language`	Target language for the rendered program. Only ‘"r"‘ is implemented; ‘"sas"‘ and ‘"python"‘ are recognised targets that fail naming every layer in this IR that has no renderer yet.
`gate`	Optional approval gate (see ‘sign_gate()‘). When supplied, the signer, moment and reason are reproduced in the program header so a reader of the code alone can see who approved it. The header states plainly that an approval is not evidence of correctness; omitting the gate leaves the rendered program byte-identical to an ungated render.

Details

Render a full analysis program from an IR

Value

A single character string of R code.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
program <- render_program(ir)
cat(substr(program, 1, 200))
```

---

render_variable *Render a variable*

Description

Renders a variable's full step pipeline via [render_step()], or a 'NEEDS HUMAN REVIEW' header block when the variable abstained.

Usage

```
render_variable(v)
```

Arguments

v	An 'aa_variable_ir' object.

Details

Render one variable IR as code

Value

A character string of commented R code.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
code <- render_variable(ir[[1]])
cat(code)
```

routing_plan *Dependency routing plan*

Description

Derives the execution routing directly from spec lineage instead of a hand-registered process DAG: topological waves (a variable enters wave k only when every variable it references landed in an earlier wave), per-variable dependencies, and the two fail-closed lanes - variables blocked by dependency cycles and variables routed to human review. The ordering is computed, not declared; [validate_ir()] refuses cyclic specs outright, so a plan built from a validated IR always exists.

Usage

```
routing_plan(ir, known_columns = character())
```

Arguments

ir	List of ‘aa_variable_ir‘ objects.
known_columns	Character vector of column names assumed to pre-exist in the source data (e.g. SDTM base columns).

Details

Plan the derivation routing for an IR

Value

A data.frame with columns ‘variable‘, ‘wave‘ (0-based topological level; ‘NA‘ for cycle-blocked or human-routed variables), ‘depends_on‘ (comma-joined variable names produced earlier in the same IR), ‘layers‘ (layer chain or ‘"needs_human"‘), and ‘needs_human‘. Attributes: ‘blocked‘ (cycle-blocked variables) and ‘duplicate‘ (columns produced by more than one variable).

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
plan <- routing_plan(ir)
plan[, c("variable", "wave", "layers")]
```

run_agreement *Cross-tabulate classification backends at every fingerprint level*

Description

Classifies the eval corpus once per voter and cross-tabulates the resulting IRs at all three [ir_fingerprint()] levels (‘abstention‘, ‘layer_chain‘, ‘full‘).

What this measures - and what it does not. Agreement is computed pre-execution over voters that all read the SAME spec text: it measures translation consistency and prompt variance, not correctness, and it is NOT independent double programming (every voter shares one spec, one layer registry, and one IR schema). The corpus it runs on is a self-scoring regression corpus - its expected values were generated by the ‘rules‘ backend itself - not an oracle, so voter agreement is agreement between translation strategies, never evidence that any translation is right. For LLM voters to say anything meaningful they should differ in model family AND in prompt variant: two voters sharing one convention-asserting prompt measure prompt compliance, not agreement.

Fails loudly, naming every credential environment variable it looked for, when an LLM voter cannot actually be run - an unmeasured result must never be readable as a measured one.

Usage

```
run_agreement(voters, dataset = "ADSL")
```

Arguments

voters	Named list of voters. Each is either a backend name (‘"rules"‘, ‘"llm"‘) or a list with ‘backend‘ and optionally ‘chat‘, ‘prompt_variant‘, ‘samples‘, ‘consensus‘. LLM voters should differ in model family AND prompt variant (see description).
dataset	Dataset tag of the corpus cases to run (default ‘"ADSL"‘).

Value

An ‘aa_agreement‘ list with ‘dataset‘, ‘voters‘, ‘per_case‘ (one row per case per level with each voter’s fingerprint and an ‘agree‘ flag), ‘rates‘ (agreement rate per level) and ‘crosstab‘ (per level, a contingency table of the first two voters’ human-readable labels). Printed via [print.aa_agreement()].

Examples

```
# Two rules voters agree trivially (same deterministic translator):
res <- run_agreement(list(a = "rules", b = "rules"), dataset = "ADSL")
print(res)
```

run_evals *Run the spec-to-IR classification evals*

Description

Executes every corpus case for one dataset: builds a one-row spec data frame per case, classifies it with [classify_variables()], and compares the produced step-layer sequence (‘"needs_human"‘ when the classifier abstains) against the expected sequence. Prints a summary and returns the per-case results invisibly. This is the package’s production-consistency regression gate: the ‘rules‘ backend must score accuracy 1.0 against the corpus.

Usage

```
run_evals(backend = c("rules", "llm"), chat = NULL, dataset = "ADSL")
```

Arguments

backend	Classifier backend, ‘"rules"‘ (default, deterministic, offline) or ‘"llm"‘ (requires ‘chat‘).
chat	ellmer chat object; required when ‘backend = "llm"‘, ignored otherwise.
dataset	Dataset tag of the corpus cases to run (default ‘"ADSL"‘).

Value

A data.frame (invisibly) with columns ‘id‘, ‘variable‘, ‘expected‘, ‘got‘, ‘ok‘.

Examples

```
res <- run_evals(backend = "rules", dataset = "ADSL")
evals_accuracy(res)
```

run_validation *Run validation*

Description

Executes every check attached to each variable’s layers against the resulting dataset and reports PASS/FAIL/MANUAL per check. ‘needs_human‘ variables report ‘MANUAL‘. This is the deterministic validation gate paired with generated code.

Usage

```
run_validation(data, ir, quiet = FALSE)
```

Arguments

data	The analysis dataset produced by executing the rendered code.
ir	List of ‘aa_variable_ir‘ objects used to derive ‘data‘.
quiet	If ‘TRUE‘ (default) the status table is not printed.

Details

Run deterministic validation checks on executed data

Value

Invisibly, a data.frame with columns ‘variable‘, ‘check‘, ‘status‘, ‘details‘.

Examples

```
ir <- list(new_variable_ir("ADSL", "STUDYID",
  steps = list(new_step("assign", list(target = "STUDYID", literal = "STUDY1")))
))
res <- run_validation(data.frame(STUDYID = "STUDY1"), ir)
res
```

---

spec_variables *Spec rows for a dataset*

Description

Subsets a spec data frame to the rows of one dataset.

Usage

```
spec_variables(spec, dataset)
```

Arguments

spec	Spec data.frame compatible with [read_spec_df()].
dataset	Dataset name to subset to (e.g. ‘"ADSL"‘).

Details

Extract one dataset’s spec rows

Value

The subset spec data frame; errors when no rows match.

Examples

```
rows <- spec_variables(mock_spec_adsl(), "ADSL")
nrow(rows)
```

validate_ir *IR validation gate*

Description

Runs the schema and semantic gate every IR must pass before rendering: class/shape, layer known, required args present, exclusive-arg groups, per-layer semantic rules (enum values, formula token whitelists, label/break length agreement), BDS-artifact assign guards, vector-arg typing, foreign-only dataset detection, and column-rename tracking: a step that references a column on a dataset where an earlier merge brought that column in under a different name is reported (the old name only exists in the merge's source dataset). Dependency cycles are gated at two levels: variable-level cycles report 'cyclic dependencies: ...', and cycles between deliverable datasets report a distinct 'cyclic deliverable dependencies: ...' problem naming the datasets that cannot be built in any order. Problems are reported, never guessed or repaired. String arguments are limited to 16384 bytes. Expressions admit uppercase identifiers, finite numbers, comparison and logical operators, arithmetic, parentheses and quoted filter values. Calls, assignments, indexing, comments and statement separators are rejected.

Usage

```
validate_ir(ir)
```

Arguments

ir	List of 'aa_variable_ir' objects built by [new_variable_ir()].

Details

Validate a variable IR

Value

Character vector of human-readable problems; empty when valid.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
validate_ir(ir)
```

`write_artifact` *Write variable artifacts*

Description

Writes one '<dataset>__<variable>__<hash8>.R' artifact per variable (rendered code with CHECK comments) plus a JSON sidecar recording the IR, backend, model, validation result, and disclaimer. Same IR overwrites the same files; copies never accumulate.

Usage

```
write_artifact(
  ir,
  dir,
  backend_label = "rules",
  model = NULL,
  validation = NULL
)
```

Arguments

`ir`	List of 'aa_variable_ir' objects.
`dir`	Output directory (created when missing). No default is set: pass an explicit path such as 'tempdir()' so nothing is ever written to the user's current working directory by accident.
`backend_label`	Backend label recorded in sidecars.
`model`	Model identifier recorded in sidecars (LLM backend).
`validation`	Optional validation result recorded in sidecars.

Details

Write per-variable artifacts and sidecars

Value

Invisibly, the character vector of files written.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
out <- file.path(tempdir(), "aa-examples")
write_artifact(ir, dir = out)
list.files(out)
```

write_program_artifact *Write program artifact*

Description

Renders the full program via [render_program()], writes it as '<dataset>__program__<hash8>.R' with a JSON sidecar, and appends a 'write_program' entry to the run log.

Usage

```
write_program_artifact(
  ir,
  dir,
  backend_label = "rules",
  model = NULL,
  gate = NULL,
  require_gate = NULL
)
```

Arguments

ir	Non-empty list of 'aa_variable_ir' objects.
dir	Output directory (created when missing).
backend_label	Backend label recorded in header and sidecar.
model	Model identifier recorded in the sidecar (LLM backend).
gate	Optional approval gate from 'sign_gate()', recorded in the program header, the sidecar and the run log. A gate names a signer, a moment and a reason; it is not a claim that any derivation is correct.
require_gate	Whether a covering approval gate is mandatory. 'NULL' (default) reads 'getOption("admiralagent.require_gate", TRUE)': enforcement is ON out of the box, so an ungated write is refused and the refusal itself is logged. Writing ungated requires an explicit opt-out - 'require_gate = FALSE' on the call, or the option switched off for the session - and is never silent: the rendered program carries an 'UNGATED DRAFT' banner, the sidecar and the audit record carry 'release_grade = "ungated-draft"' and 'gate_enforced = FALSE', and the first ungated write of a session prints a notice. An ungated artifact is never release-grade.

Details

Write the assembled program artifact

Value

Invisibly, the character vector of files written.

Examples

```
spec <- mock_spec_adsl()
ir <- classify_variables(spec, "ADSL", backend = "rules")
out <- file.path(tempdir(), "aa-examples-program")
write_program_artifact(ir, dir = out, require_gate = FALSE)
list.files(out)
```

Index

aa_layers, 3
accept_model, 4
agreement_overstatement, 5
artifact_hash, 6

build_context, 6
build_system_prompt, 7

check_admiral_compat, 8
classify_variables, 8
classify_variables_dual, 10
classify_variables_llm, 11
codelist_from_data, 12

dataset_json_meta, 13
dual_disagreements, 14

evals_accuracy, 14
execute_ir, 15

ir_dependency_report, 16

layer_docs, 17
layer_names, 18
load_evals, 18

mcp_handle_request, 19
mcp_serve, 20
mcp_tool_specs, 20
mock_metacore, 21
mock_spec_adsl, 22

new_step, 22
new_variable_ir, 23

print.aa_agreement, 24
print.aa_model_acceptance, 24

read_artifact, 25
read_dataset_json, 26
read_define, 27
read_spec, 28
read_spec_df, 29
render_program, 30
render_variable, 31
routing_plan, 32
run_agreement, 33
run_evals, 34
run_validation, 34

spec_variables, 35

validate_ir, 36

write_artifact, 37
write_program_artifact, 38