

Package: finnts (via r-universe)

October 29, 2026

Title Microsoft Finance Time Series Forecasting Framework

Version 0.7.0.9018

Description Automated time series forecasting developed by Microsoft Finance. The Microsoft Finance Time Series Forecasting Framework, aka Finn, can be used to forecast any component of the income statement, balance sheet, or any other area of interest by finance. Any numerical quantity over time, Finn can be used to forecast it. While it can be applied outside of the finance domain, Finn was built to meet the needs of financial analysts to better forecast their businesses within a company, and has a lot of built in features that are specific to the needs of financial forecasters. Happy forecasting!

URL <https://microsoft.github.io/finnts/>,
<https://github.com/microsoft/finnts>

BugReports <https://github.com/microsoft/finnts/issues>

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Additional_repositories <https://bgreenwell.r-universe.dev>

Imports callr, cli, Cubist, dials (>= 1.3.0), digest, doParallel, dplyr, earth, feasts, foreach, forecast, fs, generics, glue, glmnet, gtools, httr, hts, jsonlite, kernlab, lubridate, magrittr, methods, parallel, parsnip, plyr, purrr, recipes, rlang, rsample, rules, snakecase, stringr, tibble, tidyr, tidyselect, timetk, tune, vroom, workflows

Suggests arrow (>= 8.0.0), AzureStor, Boruta (>= 8.0.0), caret, corrr, ellmer (>= 0.4.0), energy, knitr, Microsoft365R, nixtlar, notebookutils, processx (>= 3.9.0), qs2, ranger, reactable, rmarkdown, S7, sparklyr, testthat (>= 3.0.0), tseries, withr, xgboost, vip (>= 0.5.0)

Config/testthat/edition 3

Depends R (>= 4.0), modeltime

VignetteBuilder knitr

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

Repository <https://microsoft.r-universe.dev>

Date/Publication 2026-10-29 16:19:15 UTC

RemoteUrl <https://github.com/microsoft/finnts>

RemoteRef HEAD

RemoteSha 3e036efd2f90e38a8916372e970681cb9a78002d

Contents

ask_agent	3
chat_copilot	4
create_custom_model	5
custom_model_control	9
custom_model_examples_template	10
custom_model_feedback	11
detect_hierarchy	12
ensemble_models	13
final_models	15
forecast_time_series	18
get_agent_forecast	23
get_best_agent_run	24
get_eda_data	26
get_forecast_data	27
get_prepped_data	28
get_prepped_models	29
get_run_info	30
get_summarized_models	31
get_trained_models	32
iterate_forecast	33
list_models	36
prep_data	36
prep_models	39
print.finnts_custom_model	42
set_agent_info	43
set_project_info	45
set_run_info	47
train_models	48
update_forecast	50
validate_prepared_input	54

Index

58

`ask_agent`*Ask Questions About Finn Agent Forecast Results*

Description

This function allows users to ask questions about their Finn AI Agent forecast results and get answers based on the outputs from `iterate_forecast()` or `update_forecast()`. It uses an LLM-driven workflow to generate and execute R code to answer questions.

Usage

```
ask_agent(agent_info, question)
```

Arguments

<code>agent_info</code>	Agent info from <code>set_agent_info()</code>
<code>question</code>	A character string containing the question to ask about the forecast

Value

A character string containing the answer to the question

Examples

```
## Not run:
# After running iterate_forecast() or update_forecast()

# Ask about exploratory data analysis
answer <- ask_agent(
  agent_info = agent_info,
  question = "Were there any missing values in the data?"
)

# Ask about forecast accuracy
answer <- ask_agent(
  agent_info = agent_info,
  question = "What is the average weighted MAPE across all time series?"
)

# Ask about models used
answer <- ask_agent(
  agent_info = agent_info,
  question = "Which models were used for the forecast?"
)

# Ask about feature importance
answer <- ask_agent(
  agent_info = agent_info,
  question = "What are the top 5 most important features in the xgboost model?"
```

```

)

# Ask about specific time series
answer <- ask_agent(
  agent_info = agent_info,
  question = "What is the forecast for product XYZ for the next 3 months?"
)

## End(Not run)

```

chat_copilot

Use GitHub Copilot CLI as a Finn LLM

Description

Creates a serializable configuration template for isolated Finn Agent sessions. Unlike an ellmer Chat, this adapter replays system instructions and conversation history as text on each request. Finn remains responsible for tools, proposal validation, and retries.

Usage

```
chat_copilot(model = "auto", command = "copilot", timeout = 120)
```

Arguments

model	A nonempty Copilot model identifier available to your account. Defaults to "auto", letting Copilot select the model for each request.
command	Copilot executable name or path. Defaults to "copilot". On Windows, prefer the standalone copilot.exe rather than an npm .cmd shim.
timeout	Positive finite timeout in seconds for each CLI request.

Details

Requires GitHub Copilot CLI 1.0.93 or later, an entitled GitHub account, and the optional **processx** package. Authentication uses COPILOT_GITHUB_TOKEN, GH_TOKEN, or GITHUB_TOKEN (in that order), or credentials retrieved privately with gh auth token. Tokens are resolved in the process making the request, not stored in the template.

Every request uses a new CLI home and working directory under tempdir(). The CLI saves local configuration, transcripts, and session state there; these can contain sensitive prompt and response data. Only use data approved for GitHub Copilot. Repository instructions, hooks, IDE connections, built-in MCP servers, and model tools are disabled. Cached Copilot-only logins are not copied into the isolated home; use an environment token or gh auth login instead.

Use this template with `create_custom_model()` as well as Agent workflows. Custom-model authoring also requires ellmer >= 0.4.0 and S7 for response types. The adapter requests JSON and checks its shape locally; Copilot CLI does not enforce the schema. Invalid responses raise finnts_copilot_structured_error without committing history or retrying. The usual custom-model approval and numerical validation gates still apply.

Value

A `finnts_copilot_chat` environment with `$chat(prompt, echo = FALSE)`, `$chat_structured(prompt, type, echo = $set_system_prompt(prompt), $get_system_prompt(), $get_model(), and $get_turns()` methods. `$chat()` returns one character string. `$chat_structured()` accepts `ellmer` basic, `enum`, `array` and `object` types and returns parsed JSON without vector/data-frame simplification. It supports only `convert = FALSE`; `echo` is `"none"`, `"text"`, `"output"` or `"all"`. Raw JSON Schema and ignored-field types are not supported. Successful exchanges update in-memory history; failed requests do not.

Examples

```
## Not run:
llm <- chat_copilot()
agent <- set_agent_info(
  project_info = project, llm = llm,
  input_data = hist_data, forecast_horizon = 6
)

## End(Not run)
```

<code>create_custom_model</code>	<i>Create a Validated Custom Model</i>
----------------------------------	--

Description

Turn plain-language business logic into a reusable custom model. Use `custom_model_examples_template()` to write worked examples of the forecasts you expect, and `custom_model_control()` for advanced settings.

Usage

```
create_custom_model(
  instructions = NULL,
  llm = NULL,
  input_data = NULL,
  run_info = NULL,
  combo_variables = NULL,
  target_variable = NULL,
  date_type = NULL,
  forecast_horizon = NULL,
  external_regressors = NULL,
  hist_start_date = NULL,
  hist_end_date = NULL,
  name = NULL,
  model_type = NULL,
  validation_examples = NULL,
  approval = "manual",
  control = NULL
)
```

Arguments

instructions	Nonblank business logic for the new model.
llm	An ellmer Chat object or <code>chat_copilot()</code> template. Generation uses a fresh, empty-history session. Both providers require ellmer $\geq$ 0.4.0 for structured response types; Copilot also uses S7 and its CLI dependencies.
input_data	Local data frame containing a column named Date of class Date, the target, series keys and declared drivers. Supply this or run_info.
run_info	Compatible prepared run that includes the raw C1 recipe, read without changes. Its metadata is inherited; explicitly supplied metadata must agree with it.
combo_variables	Series-key column names. For raw input, omission is allowed only with unique dates and no varying undeclared non-driver columns.
target_variable	Numeric target column name; required for raw input.
date_type	One of "day", "week", "month", "quarter", "year". Omission requires at least three actual dates per series matching exactly one shared complete cadence. Fiscal calendars are not inferred.
forecast_horizon	Positive integer periods to forecast; required for raw input.
external_regressors	Explicit driver column names; otherwise inherited from prepared metadata or empty. Drivers are never selected automatically. Include both historical-only and future-required drivers. Specify their intended availability and any lag or internal generation rule in instructions; the reviewed contract freezes that declaration.
hist_start_date, hist_end_date	Optional inclusive Date period labels. For first-of-month data, August 2026 is <code>as.Date("2026-08-01")</code> . Bounds select authoring history, not a permanent cutoff for later model fits.
name	Optional safe model name, preserved in the confirmed rule.
model_type	Unique subset of <code>c("local", "global")</code> . Defaults to local; pooled/global behavior requires explicit compatible settings.
validation_examples	Recommended data frame of two to eight independently calculated worked examples, including at least one edge case, in the format returned by <code>custom_model_examples_template()</code> . Its expected values, errors and tolerances are mandatory during repair.
approval	"manual" (default) or "automatic". Manual approval needs an interactive R session. Automatic explicitly authorizes generated-code execution without review.
control	Advanced settings created by <code>custom_model_control()</code> . Omit for defaults.

Value

An approved, passive `finnts_custom_model` usable by `prep_models()`. It embeds no fitted state, training data or Chat. Cancellation and failed gates raise errors; use `custom_model_feedback()` on a returned model or captured error for bounded status and next-action details.

Inputs and Validation

Every series needs finite targets on a complete common historical grid. Every declared driver must cover fitting history. Drivers also require values on every future forecast date unless explicitly declared historical-only in the reviewed contract. Missing future cells never imply historical-only use. Historical-only values are removed from prediction input even when supplied. Use observed lags/fitted statistics or explicitly specify how the model generates later driver values, including alignment, starting state and order. Unresolved business choices block source execution for clarification. Gaps and duplicate keys fail; authoring does not fill or infer business values. Existing C1 historical zero-fill behavior is unchanged; a prepared zero does not retain evidence that the original historical value was missing. Without `hist_end_date`, series must share their last finite target date. This does not certify business close: specify a cutoff when later rows contain partial actuals, plans or prior forecasts. Driver holdout scores are conditional on supplied values; historical plan vintages/publication lags are not verified.

Raw input is privately prepared; existing runs and Chat objects are unchanged. Prepared runs must include the raw C1 recipe; built-in transformations such as differencing, Box-Cox, outlier cleaning and multistep never alter C1. Local sampling retains up to three series or raw hierarchy leaves. Explicit global calls validate a deterministic representative sample of at most 100 series: extreme-scale, zero-heavy, negative, flattest and shortest series are always kept, multi-variable combo levels are covered when they fit, and remaining series fill in hash order. Every series still passes the full-panel input checks, and forecasting fits the authored model on the complete cohort. All paths retain at most 10000 historical node rows. Prepared hierarchies retain all nodes. Hierarchy rules must be local and univariate; forecast execution requires a mixed custom/FinnTS pool and reconciliation. New global contracts separately declare per-series, per-group or full-cohort prediction. Required peers and complete horizons are checked before prediction. The LLM still writes pooling, regression, transformations and allocation logic; portable calendar/key helpers and prompts do not choose formulas or fit models.

Approved envelopes are reusable by default: save the complete returned object with `saveRDS()`, read it back and supply it to later standard or Agent runs. The authoring horizon describes validation examples, not a future horizon whitelist. Each fresh fit uses its current horizon, history, cutoff and drivers. Positive whole-number horizons are supported subject to the fixed rule's history, driver-type, cadence, mode and business constraints. Finite validation cases do not prove correctness at every horizon. Same-run restarts still require unchanged inputs/settings; reuse does not advance an old fitted object.

Examples must forecast the next complete horizon immediately after their own history and include enough history for the rule. Supplied answers are never computed from the candidate or changed to make it pass. If examples are omitted, LLM-generated numerical comparisons are diagnostic: mismatches warn, while technical, required-property and coded-error checks remain mandatory. Fixed examples, chronological holdouts and fitted RDS round-trips provide evidence, not proof of arbitrary business correctness or guaranteed forecast accuracy.

Review and Execution

Manual calls show the rule, examples, candidate and execution warning, then ask `Create model?`. `yes` authorizes that exact source and conditionally approves it only if required validation passes. `no`, blank or EOF declines. `details` shows source without execution; `edit` requests a new candidate. Every repaired source version needs fresh consent. Noninteractive manual calls raise an error. Automatic mode skips manual review, not validation, and raises an error when essential business

context is unresolved.

Generated R runs in the current session with the account's filesystem/network permissions. There is no sandbox, crash isolation or hard timeout. The elapsed validation budget is checked after evaluation returns; stuck code can require restarting R. Ordinary options, RNG, working directory and library paths are restored on normal return/error, but arbitrary global or external side effects cannot be rolled back. Syntax checks and approval receipts do not authenticate a human or make arbitrary code safe. No packages are installed by authoring.

Input instructions, including accumulated review edits, are limited to 1048576 UTF-8 bytes (1 MiB), without truncation. This accommodates thousands of words and hundreds of code lines. The selected provider's context limit still applies to instructions plus schema, examples and conversation history. Separate generated-response and reviewed-source size limits are unchanged.

Instructions, examples, source and diagnostics may be confidential. Provider calls receive instructions, schema and supplied examples, not automatically actual series values. Chat objects are never saved. Feedback is advisory and never grants consent or extra retries.

Development Artifacts

This unreleased feature supports one current creation contract. Advanced settings belong only in `custom_model_control()`. Older development artifacts or argument forms must be recreated, not silently migrated.

See Also

[custom_model_examples_template\(\)](#), [custom_model_control\(\)](#), [custom_model_feedback\(\)](#), [prep_models\(\)](#). See `vignette("best-model-selection", package = "finnts")` for workflow examples, automatic defaults, protocol details and model enrollment.

Examples

```
# Worked examples: history rows hold the target, future rows hold the
# independently calculated expected forecast.
examples <- custom_model_examples_template(forecast_horizon = 2)
examples
## Not run:
model <- create_custom_model(
  instructions = "Forecast every future period as the last observed value.",
  llm = ellmer::chat_openai(),
  input_data = data.frame(Date = seq(as.Date("2021-01-01"),
    by = "month",
    length.out = 12
  ), id = "one", value = 15),
  combo_variables = "id", target_variable = "value",
  date_type = "month", forecast_horizon = 2,
  validation_examples = examples,
  approval = "automatic",
  control = custom_model_control(package_policy = "base_only")
)
custom_model_feedback(model)

## End(Not run)
```

custom_model_control *Advanced Controls for Custom Model Creation*

Description

Most callers can omit these controls. No provider calls, model execution or files are created by this constructor. Construct a new control rather than mutating one; explicit-setting markers are checked for consistency.

Usage

```
custom_model_control(
  max_attempts = 3L,
  validation_budget = 60,
  package_policy = "installed_finnts",
  required_properties = character(),
  forecast_approach = NULL
)
```

Arguments

max_attempts	Total proposals per phase, including the initial proposal; an integer from 1 through 3.
validation_budget	Elapsed validation budget in seconds, greater than zero and at most 120. Default 60; not a hard timeout or provider-call limit.
package_policy	Generated-code package boundary: "installed_finnts" (default) permits installed direct FinnTS dependencies and its existing base/recommended allowance; "base_only" permits base R. No installation.
required_properties	Mandatory checks chosen from constant_forecast, target_scale_equivariant, independent_series, fixed_window, and relative_calendar. Default none; choose only properties required by the rule.
forecast_approach	NULL inherits a prepared run, or uses "bottoms_up" for raw data. Other choices are "standard_hierarchy" and "grouped_hierarchy"; both require local, univariate rules on the raw C1 recipe and a mixed custom/FinnTS pool with reconciliation during forecast execution.

Value

A validated finnts_custom_model_control with five named settings. Invalid settings or stale explicit-setting markers raise authoring errors.

See Also

[create_custom_model\(\)](#)

Examples

```
custom_model_control()
custom_model_control(max_attempts = 2L, package_policy = "base_only")
```

custom_model_examples_template
<i>Template for Custom Model Validation Examples</i>

Description

Return a small example table in the format `create_custom_model()` expects for `validation_examples`. Each example value groups one worked case: history rows carry observed target values, future rows leave the target missing and carry the independently calculated expected forecast. Replace the illustrative values with cases calculated from your own business rule.

Usage

```
custom_model_examples_template(forecast_horizon = 3, date_type = "month")
```

Arguments

forecast_horizon	Number of future rows per example. Must match the forecast horizon used by <code>create_custom_model()</code> .
date_type	Date cadence: "year", "quarter", "month", "week" or "day". Must match the cadence used by <code>create_custom_model()</code> .

Details

The table may also contain:

- Series columns named like the forecast’s `combo_variables`. Omit them for one-series examples. An example with two or more series validates global logic; one series validates local logic.
- Driver columns named like `external_regressors`, filled on every row, including future rows.
- `edge_case` (logical). Example names starting with `edge_` are edge cases automatically.
- `expected_error`, written as "fit:code" or "predict:code", for an edge case whose model must stop with that error code. Leave expected missing on its future rows.
- `how`, a short description of how the expected values were calculated, and `tolerance`, an absolute tolerance up to 0.01.

The target column may be named `Target` or like the forecast’s `target_variable`. Supply two to eight examples including at least one edge case. Examples are visible to the LLM; use nonconfidential values.

Value

A data frame with columns example, Date, Target, expected and how. The template holds one normal example (last_value) and one edge case (edge_zero_history) for an illustrative "repeat the last observed value" rule.

See Also

[create_custom_model\(\)](#)

Examples

```
custom_model_examples_template()
custom_model_examples_template(forecast_horizon = 2, date_type = "quarter")
```

custom_model_feedback *Read Custom Model Creation Feedback*

Description

Obtain a machine-readable status report without generating code, granting consent, loading training data or writing files. Error conditions keep their ordinary R error semantics.

Usage

```
custom_model_feedback(x)
```

Arguments

x An approved custom model or a captured error condition from [create_custom_model\(\)](#).

Details

Status is approved, review_required, needs_input or failed. Business questions are provider proposals, not verified input errors. Error-contract details have package-generated questions and provenance: details\$error_contract identifies the reason, affected field/code, phases, model mode, example index and whether independent caller input is required. Declared errors need matching error examples and numerical controls. Conflicting business/source contracts also return needs_input. Remaining counts do not authorize retries or execution; rerun [create_custom_model\(\)](#) with clearer instructions or examples instead. Approved envelopes do not retain attempt histories, so those counts are NULL. Older models may lack resolved inputs. The accessor validates model integrity and can reject altered records. Authoring errors carry this report in feedback; other captured errors receive a generic report. Foreign or altered attached feedback is rebuilt without mutating the supplied condition. Unknown stage/policy is not invented. needs_user_input covers business questions and manual candidate review. Catching an error never converts it into an enrollable model. Private raw rows, source and arbitrary exception text are not included. Column names and provider-visible questions may still be confidential. There is no automatic upload, retry or approval.

Value

A passive version-1 list with status, stage/code, next action, bounded questions/details, candidate/request identities and remaining phase budgets. NULL counts mean unavailable historical evidence, not unlimited attempts.

See Also

[create_custom_model\(\)](#)

Examples

```
failure <- tryCatch(create_custom_model(), error = identity)
custom_model_feedback(failure)
```

detect_hierarchy	<i>Detect a standard or grouped forecasting hierarchy</i>
------------------	---

Description

Selects an approach compatible with FinnTS hierarchy construction without changing input data, running preprocessing, or saving artifacts.

Usage

```
detect_hierarchy(
  input_data,
  combo_variables,
  target_variable = NULL,
  combo_cleanup_date = NULL,
  hist_end_date = NULL
)
```

Arguments

input_data	A local data frame containing the combo columns. Repeated observations are allowed. Collect Spark data frames before calling.
combo_variables	Nonempty character vector of unique combo column names.
target_variable	Target column name, required only when combo_cleanup_date is supplied. Must contain numeric values; missing targets are allowed but infinite values are not.
combo_cleanup_date	Optional Date scalar. When supplied, apply the same inactive-series cleanup as prep_data() : retain series whose target sum between this date and hist_end_date, inclusive, is nonzero.
hist_end_date	Date scalar required for cleanup. Future target values do not contribute to the cleanup sum.

Details

Character and factor combo boundaries are trimmed using the same normalization as `prep_data()`. The caller's data is unchanged. Missing, blank, nonfinite, or unsupported combo labels, ambiguous "---"-joined combo identities, invalid cleanup inputs, and an empty retained population produce errors.

Levels are ordered by increasing distinct count, with supplied column order breaking ties, matching the engine. A standard hierarchy requires every child label to determine exactly one parent at each adjacent level and the finest level to identify every bottom-level tuple. Crossed dimensions or reused child labels select a grouped hierarchy. One retained series selects "bottoms_up" because the HTS constructors require multivariate input.

Detection describes observed relationships, not unobserved business relationships. Use the same input population and cleanup settings that will be passed to `prep_data()`. This function does not override explicit preprocessing choices. Agent setup shares the structural analysis but preserves its single-column bottoms-up policy and saved-version contracts.

Value

A list with `forecast_approach` ("standard_hierarchy", "grouped_hierarchy", or "bottoms_up"), `hierarchy_order`, named distinct counts, retained bottom-series count `total_ts`, a conflicts data frame (parent, child, child_label, parent_count), and a readable reason. Diagnostics are returned in memory only.

Examples

```
data <- data.frame(
  Region = c("North", "North", "South"),
  Site = c("A", "B", "C")
)
decision <- detect_hierarchy(data, c("Region", "Site"))
decision$forecast_approach

data$Date <- as.Date("2026-08-01")
data$Revenue <- c(10, 0, 20)
detect_hierarchy(
  data, c("Region", "Site"), target_variable = "Revenue",
  combo_cleanup_date = as.Date("2025-08-01"),
  hist_end_date = as.Date("2026-08-01")
)
```

Description

Create ensemble model forecasts

Usage

```
ensemble_models(
  run_info,
  parallel_processing = NULL,
  inner_parallel = FALSE,
  num_cores = NULL,
  seed = 123
)
```

Arguments

<code>run_info</code>	run info using the set_run_info() function
<code>parallel_processing</code>	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse.
<code>inner_parallel</code>	Run components of forecast process inside a specific time series in parallel. Can only be used if <code>parallel_processing</code> is set to NULL or 'spark'.
<code>num_cores</code>	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
<code>seed</code>	Set seed for random number generator. Numeric value.

Details

Custom-model runs validate their pinned enrollment and existing artifact identities before ensembling. Custom forecasts from the raw C1 recipe join the ensemble inputs when they pass the same backtest screening as built-in models. When C1 forecasts are present, ensembles train against original-scale actuals so raw custom and cleaned built-in targets are never averaged. Built-in-only runs retain existing behavior.

Value

Ensemble model outputs are written to disk

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    Date >= "2013-01-01",
    Date <= "2015-06-01",
    id == "M750"
  )

run_info <- set_run_info()
```

```

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3
)

prep_models(run_info,
  models_to_run = c("arima", "glmnet"),
  num_hyperparameters = 2
)

train_models(run_info,
  run_global_models = FALSE
)

ensemble_models(run_info)

```

final_models

Final Models

Description

Select Best Models and Prep Final Outputs

Usage

```

final_models(
  run_info,
  average_models = TRUE,
  max_model_average = 3,
  weekly_to_daily = TRUE,
  parallel_processing = NULL,
  inner_parallel = FALSE,
  num_cores = NULL
)

```

Arguments

run_info run info using the [set_run_info\(\)](#) function.

average_models If TRUE, create simple averages of individual models and save the eligible average selected by accuracy and future-quality checks.

max_model_average Max number of models to average together. Will create model averages for 2 models up until input value or max number of models ran.

weekly_to_daily	If TRUE, convert a week forecast down to day by evenly splitting across each day of week. Helps when aggregating up to higher temporal levels like month or quarter.
parallel_processing	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse.
inner_parallel	Run components of forecast process inside a specific time series in parallel. Can only be used if parallel_processing is set to NULL or 'spark'.
num_cores	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.

Details

Candidates are screened for complete, finite predictions and extreme future magnitudes using original-scale prepared actuals. Among eligible candidates within 0.5 percentage points or 5 percent relative weighted MAPE of the best eligible accuracy, whichever allowance is larger, selection prefers lower risk and fewer future level, trend, and seasonal concerns. Tied candidates with assessed seasonal evidence then prefer smaller amplitude distortion beyond historical cycle variation, before weighted MAPE and model identifier. This preference alone does not reject a forecast. Repeated strong historical seasonality can also support phase checks over at least three informative future periods, even when the horizon is shorter than a cycle. Short histories remain usable; unsupported seasonal checks are not assessed. Sufficient regular prepared history can support an additive or proportional trend reference when it improves both chronological historical validation blocks. Level checks then follow the projected trend and seasonal offsets, with uncertainty from historical residuals and drift variation. Proportional references use log-scale changes and seasonal comparisons only for suitable positive history. Unsupported or unstable trends retain the existing seasonal-naive or recent-median reference. Backtests retain the historical magnitude bound; supported future paths use 100 times the larger of the historical robust scale and the absolute projected reference at each step. Prepared-history imputation remains part of the evidence. These checks are engineering guardrails, not calibrated intervals or accuracy guarantees. If normal selection has no winner, complete finite candidates with usable backtest accuracy may receive a plausibility fallback. It prefers fewer failed checks, lower maximum normalized severity (including extreme magnitude), lower weighted MAPE, then stable model identity. A classed `finnts_forecast_selection_fallback` warning identifies the selected model and concerns. Missing/duplicate required keys, nonfinite predictions and unavailable accuracy cannot be relaxed; an entirely unusable pool errors. Strictly rejected components are not added to averages or ensemble training. Evaluation is deterministic for fixed inputs and creates no diagnostic files. If an individual model wins, the best eligible simple average is still saved with `Best_Model = "No"`, using the same quality-aware ranking among averages. If an average wins overall, that exact average is saved as the best model. No average artifact is required when no eligible average can be formed. Quality selection happens before hierarchical reconciliation, at each prepared hierarchy node. The selected mixture is reconciled without a second future plausibility evaluation or a whole-hierarchy replacement model. Reconciled backtests still supply reported accuracy; reconciliation does not require retained quality rankings. On retry, saved indi-

vidual and average outputs must identify one complete winner per series. A `Best_Model` column or an average filename alone is not proof of completion. Incomplete selections rebuild averages and winner flags from existing predictions without fitting models again. Complete saved winners are reused without future-quality reassessment, and every series remains in the returned result. Reconciled output is reused only when all original series have complete, unique backtest and future keys with finite forecasts, including every day of a daily-expanded week. Incomplete reconciled output is rebuilt from selected source forecasts and checked before completion is logged; storage and read errors propagate. During model-result repair of an accepted default forecast, old average outputs and selection-completion flags are not reused. Selection is rebuilt from the replacement individual predictions through the same policy, with stale averages reset using the ordinary forecast schema. This recovery does not change preparation. Older plausibility-only rejected defaults can recover from usable saved predictions and fits without training again; structurally unusable defaults remain rejected. Fresh defaults also use fallback when the normal winner has soft concerns. Ordinary update assessment and retuning do not opt into this final-selection policy. Single-series finalization reads its known log, splits and prediction paths directly, including daily-expanded weekly rows, without directory discovery. Fallback evidence is in memory only; no tracking fields are persisted. If only one individual remains eligible, the schema-correct empty optional average represents no average and remains readable during publication and restart. Empty required artifacts and malformed optional averages are errors.

Experimental custom runs validate the pinned approved pool, fitted context and saved candidate membership before selection or completion reuse. Foreign custom models, excluded models and averages containing foreign components are errors. Custom aliases remain in `Model_Name`; content hashes identify custom `Model_ID` components without conflicting with average separators. Simple averages may combine complete eligible custom, built-in and ensemble components. Ranking, accuracy allowances and plausibility policy are unchanged; custom enrollment does not guarantee that a candidate will be selected.

Explicitly reviewed custom hierarchy runs allow `average_models` and require native weekly output (`weekly_to_daily = FALSE` for weekly data). Every enrolled candidate must have complete finite node/fold/date coverage before reconciliation. No missing-node substitution or silent candidate omission is permitted. Existing hts weighting and selection remain unchanged; reconciliation may adjust base forecasts before bottom-level publication. Eligibility comes from the shared requested mixed pool, not the winners: custom models may win at every node. Custom aliases and full version IDs remain in per-model outputs; the selected mixture retains `Best-Model`.

Value

Final model outputs are written to disk.

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    Date >= "2013-01-01",
    Date <= "2015-06-01"
  )

run_info <- set_run_info()
```

```
prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3
)

prep_models(run_info,
  models_to_run = c("arima", "ets"),
  back_test_scenarios = 3
)

train_models(run_info,
  run_global_models = FALSE
)

final_models(run_info)
```

forecast_time_series *Finn Forecast Framework*

Description

Calls the Finn forecast framework to automatically forecast any historical time series.

Usage

```
forecast_time_series(
  run_info = NULL,
  input_data,
  combo_variables,
  target_variable,
  date_type,
  forecast_horizon,
  external_regressors = NULL,
  hist_start_date = NULL,
  hist_end_date = NULL,
  combo_cleanup_date = NULL,
  fiscal_year_start = 1,
  clean_missing_values = TRUE,
  clean_outliers = FALSE,
  back_test_scenarios = NULL,
  back_test_spacing = NULL,
  modeling_approach = "accuracy",
  forecast_approach = "bottoms_up",
  parallel_processing = NULL,
```

```

    inner_parallel = FALSE,
    num_cores = NULL,
    negative_forecast = FALSE,
    fourier_periods = NULL,
    lag_periods = NULL,
    rolling_window_periods = NULL,
    recipes_to_run = NULL,
    pca = NULL,
    models_to_run = NULL,
    models_not_to_run = NULL,
    run_global_models = NULL,
    run_local_models = TRUE,
    run_ensemble_models = NULL,
    average_models = TRUE,
    max_model_average = 3,
    feature_selection = FALSE,
    weekly_to_daily = TRUE,
    seed = 123,
    run_model_parallel = FALSE,
    return_data = TRUE,
    run_name = "finnts_forecast",
    custom_models = NULL,
    stationary = TRUE
  )

```

Arguments

run_info	Run info using <code>set_run_info()</code>
input_data	A data frame or tibble of historical time series data. Can also include external regressors for both historical and future data.
combo_variables	List of column headers within input data to be used to separate individual time series.
target_variable	The column header formatted as a character value within input data you want to forecast.
date_type	The date granularity of the input data. Finn accepts the following as a character string day, week, month, quarter, year.
forecast_horizon	Number of periods to forecast into the future.
external_regressors	List of column headers within input data to be used as features in multivariate models.
hist_start_date	Date value of when your input_data starts. Default of NULL is to use earliest date value in input_data.
hist_end_date	Date value of when your input_data ends. Default of NULL is to use the latest date value in input_data.

combo_cleanup_date	Date value to remove individual time series that don't contain non-zero values after that specified date. Default of NULL is to not remove any time series and attempt to forecast all of them.
fiscal_year_start	Month number of start of fiscal year of input data, aids in building out date features. Formatted as a numeric value. Default of 1 assumes fiscal year starts in January.
clean_missing_values	If TRUE, cleans missing values. Only impute values for missing data within an existing series, and does not add new values onto the beginning or end, but does provide a value of 0 for said values. Turned off when running hierarchical forecasts.
clean_outliers	If TRUE, outliers are cleaned and inputted with values more in line with historical data
back_test_scenarios	Number of specific back test folds to run when determining the best model. Default of NULL will automatically choose the number of back tests to run based on historical data size, which tries to always use a minimum of 80% of the data when training a model.
back_test_spacing	Number of periods to move back for each back test scenario. Default of NULL moves back 1 period at a time for year, quarter, and month data. Moves back 4 for week and 7 for day data.
modeling_approach	How Finn should approach your data. Current default and only option is 'accuracy'. In the future this could evolve to other areas like optimizing for interpretability over accuracy.
forecast_approach	How the forecast is created. The default of 'bottoms_up' trains models for each individual time series. 'grouped_hierarchy' creates a grouped time series to forecast at while 'standard_hierarchy' creates a more traditional hierarchical time series to forecast, both based on the hts package.
parallel_processing	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse.
inner_parallel	Run components of forecast process inside a specific time series in parallel. Can only be used if parallel_processing is set to NULL or 'spark'.
num_cores	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
negative_forecast	If TRUE, allow forecasts to dip below zero.

fourier_periods	List of values to use in creating fourier series as features. Default of NULL automatically chooses these values based on the date_type.
lag_periods	List of values to use in creating lag features. Default of NULL automatically chooses these values based on date_type.
rolling_window_periods	List of values to use in creating rolling window features. Default of NULL automatically chooses these values based on date type.
recipes_to_run	List of recipes to run on multivariate models that can run different recipes. A value of NULL runs all recipes, but only runs the R1 recipe for weekly and daily date types, and also for global models to prevent memory issues. A value of "all" runs all recipes, regardless of date type or if it's a local/global model. A list like c("R1") or c("R2") would only run models with the R1 or R2 recipe. "C1" is the raw recipe for custom models and is added automatically when custom models are selected; c("C1") alone runs only custom models.
pca	If TRUE, run principle component analysis on any lagged features to speed up model run time. Default of NULL runs PCA on day and week date types across all local multivariate models, and also for global models across all date types.
models_to_run	List of models to run. Default of NULL runs all models.
models_not_to_run	List of models not to run, overrides values in models_to_run. Default of NULL doesn't turn off any model.
run_global_models	If TRUE, run multivariate models on the entire data set (across all time series) as a global model. Can be override by models_not_to_run. Default of NULL runs global models for all date types except week and day.
run_local_models	If TRUE, run models by individual time series as local models.
run_ensemble_models	If TRUE, run ensemble models. Default of NULL runs ensemble models only for quarter and month date types.
average_models	If TRUE, create simple averages of individual models.
max_model_average	Max number of models to average together. Will create model averages for 2 models up until input value or max number of models ran.
feature_selection	Implement feature selection before model training. This requires the optional Boruta, corrr, ranger, and vip packages. vip is available from https://bgreenwell.r-universe.dev and requires R 4.1 or newer.
weekly_to_daily	If TRUE, convert a week forecast down to day by evenly splitting across each day of week. Helps when aggregating up to higher temporal levels like month or quarter.
seed	Set seed for random number generator. Numeric value.

<code>run_model_parallel</code>	If TRUE, runs model training in parallel, only works when <code>parallel_processing</code> is set to 'local_machine' or 'spark'. Recommended to use a value of FALSE and leverage <code>inner_parallel</code> for new features.
<code>return_data</code>	If TRUE, return the forecast results. Used to be backwards compatible with previous finnts versions. Recommended to use a value of FALSE and leverage <code>get_forecast_data()</code> for new features.
<code>run_name</code>	Name used when submitting jobs to external compute like Azure Batch. Formatted as a character string.
<code>custom_models</code>	Named list of experimental approved custom-model envelopes, enrolled through <code>models_to_run</code> . A NULL <code>models_to_run</code> runs all built-in models plus every supplied custom model; an explicit <code>models_to_run</code> must name every supplied custom model, otherwise an error is raised. See <code>prep_models()</code> for the envelope contract and supported profile. Selected custom models add the raw C1 recipe to <code>recipes_to_run</code> and may run alone or alongside built-in models. Their declared drivers come from eligible original input columns, not just <code>external_regressors</code> . Supply required future values through the horizon. Incompatible explicit custom settings error before data preparation. Use <code>create_custom_model()</code> for reviewed authoring.
<code>stationary</code>	Apply differencing during preparation; retains the existing TRUE default. Applies only to built-in R1/R2 recipes; custom models always use the raw, original-scale C1 recipe.

Details

Final selection uses the deterministic accuracy and future-plausibility checks documented in `final_models()`. It selects from the models already produced; an invalid forecast is not repaired into a zero forecast or replaced by an additional model fit. When normal selection has no winner, a complete finite candidate may be selected with a least-issues plausibility fallback and a warning. No structurally usable candidate still results in an error. The saved nonwinning average uses the same ranking among eligible averages. Hierarchical forecasts use the selected base forecasts as input to the existing reconciliation solver. Reconciled future outputs are not evaluated to choose a replacement set or trigger additional model fitting.

Reviewed custom hierarchy models require a shared explicitly mixed custom/FinnTS pool, local-only C1 execution and no external regressors. Model averages and ensembles may include custom forecasts. Eligibility uses requested candidates, not the eventual winning models. Custom-only forecasts require a compatible bottom-up definition instead; no model is automatically added and no requested approach is changed.

Value

A list of three separate data sets: the future forecast, the back test results, and the best model per time series.

Examples

```
run_info <- set_run_info()
```

```

finn_forecast <- forecast_time_series(
  run_info = run_info,
  input_data = m750 %>% dplyr::rename(Date = date),
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3,
  back_test_scenarios = 6,
  run_model_parallel = FALSE,
  models_to_run = c("arima", "ets", "snaive"),
  return_data = FALSE
)

fcst_tbl <- get_forecast_data(run_info)

models_tbl <- get_trained_models(run_info)

```

get_agent_forecast	<i>Get the final best forecast for an agent</i>
--------------------	---

Description

This function retrieves the final forecast for a Finn agent after the forecast iteration process is complete.

Usage

```
get_agent_forecast(agent_info)
```

Arguments

`agent_info` Agent info from `set_agent_info()`

Details

For hierarchical Agent runs, returns only the final reconciled Best-Model forecast. Each series can run and select different models, recipes, or averages during `iterate_forecast()`, so the selected forecasts are reconciled together rather than producing a comparison hierarchy for every model. Best-Model does not mean the same model family won for every series. This best-only reconciled output contract also applies after `update_forecast()`. It does not change non-hierarchical candidate output.

For non-agentic hierarchical runs, `get_forecast_data()` returns all successfully saved per-model reconciled forecasts plus Best-Model.

The final forecast artifact must exist. Missing local artifacts raise a required-artifact error; storage-provider and deserialization errors propagate rather than returning an empty forecast. Reads use the exact artifact path without directory discovery and do not modify saved outputs.

Value

A tibble containing the final forecast for the agent.

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")

# set up agent info
agent_info <- set_agent_info(
  project_info = project,
  llm = llm,
  input_data = hist_data,
  forecast_horizon = 6
)

# run the forecast iteration process
iterate_forecast(
  agent_info = agent_info,
  max_iter = 3,
  weighted_mape_goal = 0.03
)

# get the final forecast for the agent
final_forecast <- get_agent_forecast(agent_info = agent_info)

## End(Not run)
```

get_best_agent_run	<i>Get the best run for an agent</i>
--------------------	--------------------------------------

Description

This function retrieves the best run information for a Finn agent after the forecast iteration process is complete.

Usage

```
get_best_agent_run(agent_info)
```

Arguments

agent_info Agent info from set_agent_info()

Value

A tibble containing the best run information for the agent.

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")

# set up agent info
agent_info <- set_agent_info(
  project_info = project,
  llm = llm,
  input_data = hist_data,
  forecast_horizon = 6
)

# run the forecast iteration process
iterate_forecast(
  agent_info = agent_info,
  max_iter = 3,
  weighted_mape_goal = 0.03
)

# get the best run information for the agent
best_run_info <- get_best_agent_run(agent_info = agent_info)

## End(Not run)
```

`get_eda_data`*Get EDA Data*

Description

Load exploratory data analysis results from a Finn Agent run and return as a single data frame

Usage

```
get_eda_data(agent_info)
```

Arguments

`agent_info` Agent info from `set_agent_info()`

Value

A data frame containing all EDA results with columns:

- `Combo`: Time series identifier
- `Analysis_Type`: Type of EDA analysis (e.g., "ACF", "PACF", "Stationarity", etc.)
- `Metric`: Specific metric or measure within each analysis type
- `Value`: Numeric or character value of the metric

Examples

```
## Not run:  
# Get EDA results for all time series  
eda_df <- get_eda_data(agent_info)  
  
# Filter for specific analysis types  
acf_results <- eda_df %>%  
  dplyr::filter(Analysis_Type == "ACF")  
  
# Filter for specific time series  
ts_results <- eda_df %>%  
  dplyr::filter(Combo == "Product_A--Region_1")  
  
## End(Not run)
```

get_forecast_data	<i>Get Final Forecast Data</i>
-------------------	--------------------------------

Description

Get Final Forecast Data

Usage

```
get_forecast_data(run_info, return_type = "df")
```

Arguments

run_info	run info using the set_run_info() function
return_type	return type

Details

For non-agentic standard or grouped hierarchical runs, returns every successfully saved per-model reconciled forecast together with Best-Model. Filter Best_Model == "Yes" to retain only the reconciled selected forecast. Best-Model can combine different winning models or averages across series; it does not require one model family to win everywhere. Models that did not successfully produce a reconciled artifact are not added to the output.

Use [get_agent_forecast\(\)](#) for final Agent results. Hierarchical Agent output contains only the reconciled selected forecast, because [iterate_forecast\(\)](#) can run different model and recipe sets for different series. It is not an all-model reconciled comparison table.

Value

table of final forecast results

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    id == "M2",
    Date >= "2012-01-01",
    Date <= "2015-06-01"
  )

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
```

```

    date_type = "month",
    forecast_horizon = 3,
    recipes_to_run = "R1"
  )

  prep_models(run_info,
    models_to_run = c("arima", "ets"),
    num_hyperparameters = 1
  )

  train_models(run_info,
    run_local_models = TRUE
  )

  final_models(run_info,
    average_models = FALSE
  )

  fcst_tbl <- get_forecast_data(run_info)

```

get_prepped_data

*Get Prepped Data***Description**

Get Prepped Data

Usage

```
get_prepped_data(run_info, recipe, return_type = "df")
```

Arguments

run_info	run info using the set_run_info() function
recipe	recipe to return. Either a value of "R1" or "R2"
return_type	return type

Value

table of prepped data

Examples

```

data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    id == "M2",

```

```
      Date >= "2012-01-01",
      Date <= "2015-06-01"
    )

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3,
  recipes_to_run = "R1"
)

R1_prepped_data_tbl <- get_prepped_data(run_info,
  recipe = "R1"
)
```

get_prepped_models	<i>Get Prepped Model Info</i>
--------------------	-------------------------------

Description

Get Prepped Model Info

Usage

```
get_prepped_models(run_info)
```

Arguments

run_info run info using the [set_run_info\(\)](#) function

Value

table with data related to model workflows, hyperparameters, and back testing

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    id == "M2",
    Date >= "2012-01-01",
    Date <= "2015-06-01"
  )
```

```

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3,
  recipes_to_run = "R1"
)

prep_models(run_info,
  models_to_run = c("arima", "ets"),
  num_hyperparameters = 1
)

prepped_models_tbl <- get_prepped_models(run_info = run_info)

```

get_run_info

Get run info

Description

Lets you get all of the logging associated with a specific project or run.

Usage

```

get_run_info(
  project_name = NULL,
  run_name = NULL,
  storage_object = NULL,
  path = NULL
)

```

Arguments

project_name	Name used to group similar runs under a single project name.
run_name	Name to distinguish one run of Finn from another. The current time in UTC is appended to the run name to ensure a unique run name is created.
storage_object	Used to store outputs during a run to other storage services in Azure. Could be a storage container object from the 'AzureStor' package to connect to ADLS blob storage or a OneDrive/SharePoint object from the 'Microsoft365R' package to connect to a OneDrive folder or SharePoint site. Default of NULL will save outputs to the local file system.
path	String showing what file path the outputs should be written to. Default of NULL will write the outputs to a temporary directory within R, which will delete itself after the R session closes.

Value

Data frame of run log information

Examples

```
run_info <- set_run_info(  
  project_name = "finn_forecast",  
  run_name = "test_run"  
)  
  
run_info_tbl <- get_run_info(  
  project_name = "finn_forecast"  
)
```

`get_summarized_models` *Get the trained model summaries info for an agent*

Description

This function retrieves the final summarized model info (hyperparameters, recipe steps, feature importance, etc.) after an agent completes its run. Built-in variable-importance rows are included when the optional vip package was available while summaries were generated; all other summary sections remain available without it. Custom models expose their exact version, confirmed interpretation and representation without source text or arbitrary fitted-state introspection. Driver importance is the mean signed increase in historical holdout RMSE under ten deterministic cyclic permutations, conditional on supplied drivers. It is not causal attribution or a percentage. Generating these summaries refits trusted custom source once per fitted workflow on earlier history and predicts complete holdout cohorts; no-driver models need no diagnostic fit. Non-estimable cases are disclosed through warnings and method/status rows.

Usage

```
get_summarized_models(agent_info)
```

Arguments

`agent_info` Agent info from `set_agent_info()`

Value

A tibble containing the summarized models for the agent. Built-in importance sections may be omitted without vip. Custom importance does not require vip. Existing saved summaries are not migrated; newly generated summaries can contain additional importance metadata rows.

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")

# set up agent info
agent_info <- set_agent_info(
  project_info = project,
  llm = llm,
  input_data = hist_data,
  forecast_horizon = 6
)

# run the forecast iteration process
iterate_forecast(
  agent_info = agent_info,
  max_iter = 3,
  weighted_mape_goal = 0.03
)

# get the final model summaries for an agent
model_summary <- get_summarized_models(agent_info = agent_info)

## End(Not run)
```

get_trained_models	<i>Get Final Trained Models</i>
--------------------	---------------------------------

Description

Get Final Trained Models

Usage

```
get_trained_models(run_info)
```

Arguments

run_info run info using the `set_run_info()` function

Value

table of final trained models

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    id == "M2",
    Date >= "2012-01-01",
    Date <= "2015-06-01"
  )

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3,
  recipes_to_run = "R1"
)

prep_models(run_info,
  models_to_run = c("arima", "ets"),
  num_hyperparameters = 1
)

train_models(run_info,
  run_global_models = FALSE,
  run_local_models = TRUE
)

final_models(run_info,
  average_models = FALSE
)

models_tbl <- get_trained_models(run_info)
```

Description

This function orchestrates the forecast iteration process for a Finn agent, including exploratory data analysis,

Usage

```
iterate_forecast(
  agent_info,
  max_iter = 3,
  weighted_mape_goal = 0.03,
  parallel_processing = NULL,
  inner_parallel = FALSE,
  num_cores = NULL,
  seed = 123
)
```

Arguments

<code>agent_info</code>	Agent info from <code>set_agent_info()</code>
<code>max_iter</code>	Maximum number of iterations for forecast optimization. This is a total per search (the global search and each time series' local search) for the Agent version, not a number of extra iterations: calling again on the same Agent run with a higher value runs only the remaining iterations, and a lower value runs nothing new.
<code>weighted_mape_goal</code>	Weighted MAPE goal the agent is trying to achieve for each time series. Series that already meet it are not searched again, and a search can stop early once it is reached. On the same Agent run, a stricter goal reopens time series whose best model is global and that miss the new goal.
<code>parallel_processing</code>	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse. Ellmer-backed workflows require ellmer 0.4.0 or later on the main process and every worker. Copilot-backed workflows require the CLI, processx, and authentication on each worker; see chat_copilot() .
<code>inner_parallel</code>	Run components of forecast process inside a specific time series in parallel. Can only be used if <code>parallel_processing</code> is set to NULL or 'spark'.
<code>num_cores</code>	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
<code>seed</code>	Set seed for random number generator. Numeric value.

Details

Future quality is evaluated when `final_models()` selects the winner within each iteration, including usable least-issues fallbacks with warnings. Parallel fallback warnings are relayed to the caller

without saved fields. Iteration ranking starts from the earliest minimum WMAPE and may prefer a later eligible iteration within 10 percent relative WMAPE when its average model WMAPE is strictly lower. Local mean, median, and standard deviation summarize individual-model backtests, excluding simple averages. This retains evidence that a setting improves other models even if the current best model does not improve. Global summary fields retain their original meaning: run WMAPE for mean and median, and zero standard deviation. Agent comparisons and stopping use WMAPE rounded to four decimal places. Search-context selection does not overwrite a better saved local forecast. Global promotion applies to one complete iteration, and all saved global winners must reference that same iteration. Individual models and averages may differ within it. Mixed global iteration metadata is rejected before publication or update; interrupted writes are not treated as successful. Normal accuracy-goal stopping requires a complete eligible result and finite WMAPE, not another soft-quality check. A winner with soft concerns may therefore beat an earlier winner on accuracy. Loaded backtests, run history, and best-run metrics are reused without re-evaluating past future paths. Rejected evaluations still consume iteration budget without repeating the same fit as an infrastructure retry. If no eligible result exists, the workflow fails or an enabled local phase handles unresolved global series. Rankings are kept in memory, not new files. Hierarchical accuracy uses reconciled backtests; later comparisons and reconciliation do not require retained source-node quality rankings. Final reconciliation publishes the selected mixture without future-quality scoring, whole-set fallback, or extra refitting.

Custom pools enrolled by `set_agent_info()` constrain every proposal to approved models and original-scale R1 settings, plus the raw C1 recipe. Missing proposal fields do not acquire built-in defaults. Local/mounted CSV data, RDS objects and sequential, local-machine or Spark execution are required; `inner_parallel` applies to custom models as it does to built-in models. Global-only pools require at least two series. Saved parent and child identities are verified before completed results can be reused. Ordinary EDA still runs; model source and approval payloads are not sent to forecast reasoning. The Agent can choose a subset but cannot rewrite source, calibrate fixed parameters or add models. Generic summaries preserve exact custom identity without executing source. Compatible new-data replay uses `update_forecast()` with explicit `allow_iterate_forecast = FALSE` and a new identically enrolled parent. Changed enrollment or series requires iteration.

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")
```

```
# set up agent info
agent_info <- set_agent_info(
  project_info = project,
  llm = llm,
  input_data = hist_data,
  forecast_horizon = 6
)

# run the forecast iteration process
iterate_forecast(
  agent_info = agent_info,
  max_iter = 3,
  weighted_mape_goal = 0.03
)

## End(Not run)
```

list_models	List all available models
-------------	---------------------------

Description

List all available models

Usage

```
list_models()
```

Value

list of models

prep_data	Prep Data
-----------	-----------

Description

Preps data with various feature engineering recipes to create features before training models

Usage

```
prep_data(
  run_info,
  input_data,
  combo_variables,
  target_variable,
  date_type,
```

```

forecast_horizon,
external_regressors = NULL,
hist_start_date = NULL,
hist_end_date = NULL,
combo_cleanup_date = NULL,
fiscal_year_start = 1,
clean_missing_values = TRUE,
clean_outliers = FALSE,
box_cox = FALSE,
stationary = TRUE,
forecast_approach = "bottoms_up",
parallel_processing = NULL,
num_cores = NULL,
fourier_periods = NULL,
lag_periods = NULL,
rolling_window_periods = NULL,
recipes_to_run = NULL,
multistep_horizon = FALSE
)

```

Arguments

run_info	Run info using set_run_info()
input_data	A standard data frame, tibble, or spark data frame using sparklyr of historical time series data. Can also include external regressors for both historical and future data. Leading and trailing whitespace in character combo-variable values is removed before Finn creates internal series identifiers.
combo_variables	List of column headers within input data to be used to separate individual time series.
target_variable	The column header formatted as a character value within input data you want to forecast.
date_type	The date granularity of the input data. Finn accepts the following as a character string: day, week, month, quarter, year.
forecast_horizon	Number of periods to forecast into the future.
external_regressors	List of column headers within input data to be used as features in multivariate models. Controls built-in R1/R2 recipes only. C1 retains eligible original numeric, character, factor, logical and Date columns independently of this list in bottoms_up runs; hierarchical C1 nodes keep only these aggregated regressors. Each custom workflow uses only its declared predictors plus Date and Combo.
hist_start_date	Date value of when your input_data starts. Default of NULL uses earliest date value in input_data.
hist_end_date	Date value of when your input_data ends. Default of NULL uses the latest date value in input_data.

combo_cleanup_date	Date value to remove individual time series that don't contain non-zero values after that specified date. Default of NULL is to not remove any time series and attempt to forecast all time series.
fiscal_year_start	Month number of start of fiscal year of input data, aids in building out date features. Formatted as a numeric value. Default of 1 assumes fiscal year starts in January.
clean_missing_values	If TRUE, cleans missing values. Only impute values for missing data within an existing series, and does not add new values onto the beginning or end, but does provide a value of 0 for said values.
clean_outliers	If TRUE, outliers are cleaned and inputted with values more in line with historical data.
box_cox	Apply box-cox transformation to normalize variance in data
stationary	Apply differencing to make data stationary
forecast_approach	How the forecast is created. The default of 'bottoms_up' trains models for each individual time series. Value of 'grouped_hierarchy' creates a grouped time series to forecast at while 'standard_hierarchy' creates a more traditional hierarchical time series to forecast, both based on the hts package.
parallel_processing	Default of NULL runs no parallel processing and forecasts each individual time series one after another. Value of 'local_machine' leverages all cores on current machine Finn is running on. Value of 'spark' runs time series in parallel on a spark cluster in Azure Databricks/Synapse.
num_cores	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
fourier_periods	List of values to use in creating fourier series as features. Default of NULL automatically chooses these values based on the date_type.
lag_periods	List of values to use in creating lag features. Default of NULL automatically chooses these values based on date_type.
rolling_window_periods	List of values to use in creating rolling window features. Default of NULL automatically chooses these values based on date_type.
recipes_to_run	List of recipes to run on multivariate models that can run different recipes. A value of NULL runs all recipes, but only runs the R1 recipe for weekly and daily date types. A value of "all" runs all recipes, regardless of date type. A list like c("R1") or c("R2") would only run models with the R1 or R2 recipe. Add "C1" to also write the custom-model recipe used by custom models: the padded history and future frame on the original scale, with missing values imputed only when clean_missing_values is TRUE and no outlier cleaning, Box-Cox, differencing or engineered features. C1 is never added by NULL or "all"; c("C1")

alone prepares only the custom-model recipe. Reserved outcome/assessment columns and unsupported column types are excluded. Supply complete history for every declared custom driver and future values through the forecast horizon for prediction-required drivers. Drivers explicitly declared historical-only in the reviewed custom-model contract do not require future values. Missing prediction-required values error rather than being forecast or replaced with zero. C1 source identity is pinned before preparation, including interrupted runs. Changing eligible source data requires a new `set_run_info()` run.

`multistep_horizon`

Use a multistep horizon approach when training multivariate models with R1 recipe.

Value

No return object. Feature engineered data is written to disk based on the output locations provided in `set_run_info()`.

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    Date >= "2013-01-01",
    Date <= "2015-06-01"
  )

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3,
  recipes_to_run = "R1"
)
```

```
prep_models
```

```
Prep Models
```

Description

Preps various aspects of run before training models. Things like train/test splits, creating hyperparameters, etc.

Usage

```

prep_models(
  run_info,
  back_test_scenarios = NULL,
  back_test_spacing = NULL,
  models_to_run = NULL,
  models_not_to_run = NULL,
  run_ensemble_models = TRUE,
  pca = NULL,
  num_hyperparameters = 10,
  seasonal_period = NULL,
  seed = 123,
  custom_models = NULL
)

```

Arguments

<code>run_info</code>	Run info using the set_run_info() function.
<code>back_test_scenarios</code>	Number of specific back test folds to run when determining the best model. Default of NULL will automatically choose the number of back tests to run based on historical data size, which tries to always use a minimum of 80% of the data when training a model.
<code>back_test_spacing</code>	Number of periods to move back for each back test scenario. Default of NULL moves back 1 period at a time for year, quarter, and month data. Moves back 4 for week and 7 for day data.
<code>models_to_run</code>	List of models to run. Default of NULL runs all models.
<code>models_not_to_run</code>	List of models not to run, overrides values in <code>models_to_run</code> . Default of NULL doesn't turn off any model.
<code>run_ensemble_models</code>	If TRUE, prep for ensemble models.
<code>pca</code>	If TRUE, run principle component analysis on any lagged features to speed up model run time. Default of NULL runs PCA on day and week date types across all local multivariate models, and also for global models across all date types.
<code>num_hyperparameters</code>	Number of hyperparameter combinations to test out on validation data for model tuning.
<code>seasonal_period</code>	One to three unique, finite numeric seasonal periods greater than 1. Custom values are used by <code>stlm-arima</code> , <code>stlm-ets</code> , and <code>tbats</code> . A value of NULL uses the defaults for <code>date_type</code> .
<code>seed</code>	Set seed for random number generator. Numeric value.

`custom_models` Named list of experimental, version-bound approved `finnts_custom_model` envelopes. A `NULL` `models_to_run` runs all built-in models plus every supplied custom model; an explicit `models_to_run` must name every supplied custom model, otherwise an error is raised. Supplied custom models cannot also appear in `models_not_to_run`. Raw definitions or registry references do not grant execution permission.

Details

Experimental custom enrollment requires named `finnts_custom_model` envelopes holding an exact passive definition plus technical validation and approval bound to its content hash (`version_id`). Validation contains `version_id`, `technical_passed = TRUE`, and a nonempty list of check-summary strings. Manual approval contains the same `version_id`, `intent_confirmed = TRUE` and `allow_code = TRUE`. Explicit automatic approval has `intent_confirmed = FALSE`, `allow_code = TRUE` and `mode = "automatic"`. Unknown fields or modes fail. These are trusted-caller attestations, not authenticated human identity, a sandbox or an approval inferred from storage. Use `create_custom_model()` for manual or explicitly automatic authoring.

Supplied custom models run when `models_to_run` is `NULL` or names them; an explicit `models_to_run` that omits one is an error. Custom models train on the original-scale C1 recipe, which must be in the `recipes_to_run` given to `prep_data()`; otherwise preparation errors. Built-in models selected in the same run use R1/R2 with any supported settings (stationary, Box-Cox, outlier cleaning, multistep, feature selection, ensembles). Built-in models on a C1-only run error, and a prepared C1 recipe without custom models is ignored with a warning. Custom forecasts can feed simple averages and ensembles; `negative_forecast = FALSE` floors custom output at zero like built-ins. Local/mounted storage, CSV/RDS/Parquet data and RDS model objects are required; sequential, local-machine and Spark execution and `inner_parallel` are supported. Custom recipes retain only Date, Combo and declared predictors; no implicit PCA, feature selection or target transformation is applied. C1 driver availability is independent of `external_regressors` in `prep_data()`. Required predictors must exist with complete supported values, including supplied future values, or preparation errors.

A definition with reviewed hierarchy capability permits `standard_hierarchy` or `grouped_hierarchy` with local univariate original-scale C1 logic. Every prepared node receives the same explicitly selected mixed custom/FinnTS pool. Custom-only hierarchy requests fail; no built-in is silently added. Forecast use requires local-only custom training, no regressors, and no weekly-to-daily expansion. Topology and exact input artifacts are pinned before enrollment.

Definitions are pinned in the normal prepared workflows, with passive attestations attached to that same artifact; no separate standard registry or pool-manifest copy is required. Reentry requires the same approved effective pool; changed definitions or preparation context require a new run. Supplying definitions without selecting them leaves built-in defaults unchanged. Registry presence alone grants neither enrollment nor execution consent. No source is evaluated during preparation and no packages are installed.

Value

Writes outputs related to model prep to disk.

Examples

```
data_tbl <- timetk::m4_monthly %>%
```

```

dplyr::rename(Date = date) %>%
dplyr::mutate(id = as.character(id)) %>%
dplyr::filter(
  Date >= "2012-01-01",
  Date <= "2015-06-01"
)

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3
)

prep_models(run_info,
  models_to_run = c("arima", "ets", "glmnet")
)

```

```
print.finnts_custom_model
```

Print a Custom Model

Description

Display a compact status summary without exposing private review contents, source code or artifact paths. Full identities remain available in the object.

Usage

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

Arguments

<code>x</code>	A validated custom model returned by create_custom_model() .
<code>...</code>	Unused print arguments.

Value

The object invisibly. Invalid model evidence raises an error.

See Also

[create_custom_model\(\)](#), [custom_model_feedback\(\)](#)

set_agent_info	<i>Set up Finn Agent Run Information</i>
----------------	--

Description

This function sets up the necessary information for a Finn Agent run, including input data, forecast horizon, and other parameters. It checks for existing runs and allows for overwriting if specified.

Usage

```
set_agent_info(
  project_info,
  llm,
  input_data,
  forecast_horizon,
  external_regressors = NULL,
  hist_end_date = NULL,
  hist_start_date = NULL,
  back_test_scenarios = NULL,
  back_test_spacing = NULL,
  combo_cleanup_date = NULL,
  allow_hierarchical_forecast = FALSE,
  negative_forecast = FALSE,
  run_global_models = NULL,
  run_local_models = TRUE,
  overwrite = FALSE,
  custom_models = NULL
)
```

Arguments

project_info	A Finn project from set_project_info()
llm	An ellmer Chat or <code>chat_copilot()</code> object used as the template for isolated agent sessions.
input_data	A data frame or tibble containing the input data. Leading and trailing white-space in character combo-variable values is removed before Finn creates internal series identifiers and writes input artifacts. Resuming a version uses previously uploaded inputs, not replacement values from this data frame. Create a new version to use revised data.
forecast_horizon	The number of periods to forecast
external_regressors	Optional character vector of candidate external regressors for built-in models. Custom models always use their declared drivers from eligible original input columns, independently of this list or the Agent's per-iteration choices.
hist_end_date	Optional Date object indicating the end of the historical data

hist_start_date	Optional Date object indicating the start of the historical data
back_test_scenarios	Optional character vector of back test scenarios
back_test_spacing	Optional numeric value for back test spacing
combo_cleanup_date	Optional Date object for combo cleanup
allow_hierarchical_forecast	Logical controlling the Agent optimization scope. TRUE expands input data to all detected hierarchy levels before optimization, runs inner iterations with forecast_approach = "bottoms_up", and reconciles the final forecast. FALSE keeps the original bottom-level series; global iterations may still compare the exact hierarchy detected by EDA after reconciling that candidate back to the bottom level.
negative_forecast	If TRUE, allow forecasts to dip below zero.
run_global_models	If TRUE, run multivariate models on the entire data set (across all time series) as a global model. Default of NULL runs global models for all date types except week and day.
run_local_models	If TRUE, run models by individual time series as local models. Default is TRUE.
overwrite	Logical indicating whether to overwrite existing agent run info
custom_models	Named approved custom-model envelopes from create_custom_model() . All supplied models are enrolled automatically alongside built-in candidates. The returned object retains names only; approved objects are saved once in an Agent-version bundle in prep_models.

Details

Custom Agent runs add the raw C1 recipe and require CSV data, RDS objects, supported local/mounted, Blob or M365 storage and no hierarchy. `negative_forecast = FALSE` floors finished custom forecasts at zero, as for built-in models. The exact approved pool is pinned before the parent run log. Existing-run setup requires equivalent explicit enrollment; changed or omitted approval requires a new run. Source is trusted R, not sandboxed. For compatible new-data updates, create a new parent with `overwrite = TRUE`, the same explicit enrollment, cadence, regressors and modes, then call [update_forecast\(\)](#) with explicit `allow_iterate_forecast = FALSE`. The positive whole-number forecast horizon may change, subject to the approved rule's history and driver requirements. Changed enrollment needs iteration. New series are routed to defaults within the existing new-series limit. The setup API has no `models_to_run` argument: Finn includes every compatible enrolled custom model from the first iteration onward, outside LLM control. Built-in candidate choices and supported preprocessing remain independent of raw C1. Custom driver values must be supplied through the forecast horizon; Finn does not forecast missing future custom drivers. The same driver declarations apply during saved-winner updates and default recovery. Changed source data requires a new Agent version (`overwrite = TRUE`), not an in-place restart. Global-only custom runs need at least two series. Each selected custom model must support an

enabled mode and the project's cadence, horizon and required predictors. The return keeps only custom-model names, not untrained models or approval payloads. The Agent-version CSV always contains one custom_models names field (NA without enrollment); the complete immutable bundle lives in prep_models, under the existing project/run/version identity. Saved approval is trusted-caller evidence, not authenticated identity; forecasting never calls the authoring LLM again.

With overwrite = FALSE, an existing version retains its saved forecast approach even if hierarchy detection changes after a package upgrade. Logged user settings must still match. New versions use current detection; one retained series or one combo column uses bottoms-up. Updating a new version across an outer forecast-approach change requires fresh iteration.

Value

A list containing the agent run information

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")

# set up agent info
agent_info <- set_agent_info(
  project_info = project,
  llm = llm,
  input_data = hist_data,
  forecast_horizon = 6
)

## End(Not run)
```

set_project_info	<i>Set up new finnts forecast project</i>
------------------	---

Description

Creates list object of information helpful in logging information about your entire forecast project.

Usage

```
set_project_info(
  project_name = "finn_project",
  path = NULL,
  combo_variables,
  target_variable,
  date_type,
  fiscal_year_start = 1,
  weekly_to_daily = TRUE,
  storage_object = NULL,
  data_output = "csv",
  object_output = "rds",
  overwrite = FALSE
)
```

Arguments

project_name	Name used to group similar runs under a single project name.
path	String showing what file path the outputs should be written to. Default of NULL will write the outputs to a temporary directory within R, which will delete itself after the R session closes.
combo_variables	Character vector of variables to combine into a combo variable.
target_variable	Character string of the target variable to forecast.
date_type	Character string of the type of date variable
fiscal_year_start	Numeric value of the month that the fiscal year starts in.
weekly_to_daily	Logical value of whether to convert weekly data to daily data. Default of FALSE will not convert weekly data to daily data.
storage_object	Used to store outputs during the project to other storage services in Azure. Could be a storage container object from the 'AzureStor' package to connect to ADLS blob storage or a OneDrive/SharePoint object from the 'Microsoft365R' package to connect to a OneDrive folder or SharePoint site. Default of NULL will save outputs to the local file system.
data_output	String value describing the file type for data outputs. Default will write data frame outputs as csv files. The other option of 'parquet' will instead write parquet files.
object_output	String value describing the file type for object outputs. Default will write object outputs like trained models as rds files. The other option of 'qs2' will instead serialize R objects as qs2 files by using the 'qs2' package.
overwrite	Logical value of whether to overwrite existing project

Value

A list of project information

Examples

```
## Not run:
project_info <- set_project_info(
  project_name = "test_project",
  combo_variables = c("Store", "Product"),
  target_variable = "Sales",
  date_type = "month"
)

## End(Not run)
```

set_run_info	<i>Set up finnts submission</i>
--------------	---------------------------------

Description

Creates list object of information helpful in logging information about your run.

Usage

```
set_run_info(
  project_name = "finn_project",
  run_name = "finn_fcst",
  storage_object = NULL,
  path = NULL,
  data_output = "csv",
  object_output = "rds",
  add_unique_id = TRUE
)
```

Arguments

project_name	Name used to group similar runs under a single project name.
run_name	Name to distinguish one run of Finn from another.
storage_object	Used to store outputs during a run to other storage services in Azure. Could be a storage container object from the 'AzureStor' package to connect to ADLS blob storage or a OneDrive/SharePoint object from the 'Microsoft365R' package to connect to a OneDrive folder or SharePoint site. Default of NULL will save outputs to the local file system.
path	String showing what file path the outputs should be written to. Default of NULL will write the outputs to a temporary directory within R, which will delete itself after the R session closes.
data_output	String value describing the file type for data outputs. Default will write data frame outputs as csv files. The other option of 'parquet' will instead write parquet files.

- object_output** String value describing the file type for object outputs. Default will write object outputs like trained models as rds files. The other option of 'qs2' will instead serialize R objects as qs2 files by using the 'qs2' package.
- add_unique_id** Add a unique id to end of run_name based on submission time. Set to FALSE to supply your own unique run name, which is helpful in multistage ML pipelines.

Value

A list of run information

Examples

```
run_info <- set_run_info(
  project_name = "test_exp",
  run_name = "test_run_1"
)
```

train_models	<i>Train Individual Models</i>
--------------	--------------------------------

Description

Train Individual Models

Usage

```
train_models(
  run_info,
  run_global_models = FALSE,
  run_local_models = TRUE,
  global_model_recipes = c("R1"),
  feature_selection = FALSE,
  negative_forecast = FALSE,
  parallel_processing = NULL,
  inner_parallel = FALSE,
  num_cores = NULL,
  seed = 123,
  debug = FALSE
)
```

Arguments

- run_info** run info using the [set_run_info\(\)](#) function
- run_global_models** If TRUE, run multivariate models on the entire data set (across all time series) as a global model. Can be override by models_not_to_run. Default of NULL runs global models for all date types except week and day.

run_local_models	If TRUE, run models by individual time series as local models.
global_model_recipes	Recipes to use in global models.
feature_selection	Implement feature selection before model training. This requires the optional Boruta, corrr, ranger, and vip packages. vip is available from https://bgreenwell.r-universe.dev and requires R 4.1 or newer.
negative_forecast	If TRUE, allow forecasts to dip below zero.
parallel_processing	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse.
inner_parallel	Run components of forecast process inside a specific time series in parallel. Can only be used if parallel_processing is set to NULL or 'spark'.
num_cores	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
seed	Set seed for random number generator. Numeric value.
debug	If TRUE, will stop on errors and show traceback.

Details

Ordinary calls retain the saved-training reuse checks. When invoked by forecast-update recovery for an unusable accepted default result, model fitting may be repeated despite an existing training log. Preparation and output formats are unchanged; the recovery control is internal and is not saved in the run log.

Selected experimental custom models use the pinned enrollment from `prep_models()`. They run only in declared enabled local/global modes, execute before built-in candidates within each mode, bypass tuning, and retain chronological analysis-fold isolation. Custom source/dependency failures and missing fold predictions are hard errors, including in mixed pools. Required predictors and finite historical targets must remain available. No source regeneration or dependency installation occurs. Custom Model_Name retains its alias; Model_ID begins with custom- and the full version hash, followed by model type and recipe. Saved workflows, fits and forecasts must match the pinned pool before reuse. Custom models train on the untransformed C1 recipe, so stationary and Box-Cox inverse steps are skipped for them and feature selection applies only to built-in models. Global custom models always use C1 regardless of global_model_recipes. Sequential, local_machine, spark and inner_parallel execution are supported. negative_forecast = FALSE floors custom forecasts at zero, as for built-in models. Existing built-in-only behavior is unchanged. Temporal custom definitions evaluate complete horizons from existing predictor rows during resampling, then retain only the original scoring rows. Analysis cutoffs, scored targets and selection rules do not change.

Reviewed custom hierarchy runs additionally require run_local_models = TRUE and run_global_models = FALSE. Each prepared hierarchy node is fitted independently under the same explicitly enrolled

mixed pool. These are base forecasts; `final_models()` applies normal reconciliation to the bottom level. Missing or changed topology, inputs, approval or candidate artifacts cannot be treated as successful saved results.

Value

trained model outputs are written to disk.

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(
    Date >= "2013-01-01",
    Date <= "2015-06-01"
  )

run_info <- set_run_info()

prep_data(run_info,
  input_data = data_tbl,
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3
)

prep_models(run_info,
  models_to_run = c("arima", "glmnet"),
  num_hyperparameters = 2,
  back_test_scenarios = 6,
  run_ensemble_models = FALSE
)

train_models(run_info)
```

update_forecast

Update Forecast with Latest Data and Inputs

Description

This function updates the forecast agent with the latest data and inputs. The default recovery behavior below applies to ordinary built-in runs; approved custom pools use the bounded replay contract described in Details. If new time series are detected in the data (up to 20\ with a floor of 10), simple forecasts are automatically created for them using default local model inputs without LLM involvement. If the number of new series exceeds the cap, an error directs the user to use `iterate_forecast()` instead.

Usage

```

update_forecast(
  agent_info,
  weighted_mape_goal = 0.1,
  allow_iterate_forecast = FALSE,
  max_iter = 3,
  parallel_processing = NULL,
  inner_parallel = FALSE,
  num_cores = NULL,
  seed = 123
)

```

Arguments

<code>agent_info</code>	Agent info from <code>set_agent_info()</code>
<code>weighted_mape_goal</code>	Weighted MAPE goal the agent is trying to achieve for each time series
<code>allow_iterate_forecast</code>	Logical indicating if the forecast iteration should be allowed if poor performance is detected, meaning >40% of time series with >20% worse weighted MAPE than previous agent run. Custom updates require this argument to be supplied explicitly as FALSE.
<code>max_iter</code>	Numeric indicating the maximum number of iterations if <code>iterate_forecast</code> is ran
<code>parallel_processing</code>	Default of NULL runs no parallel processing and forecasts each individual time series one after another. 'local_machine' leverages all cores on current machine Finn is running on. 'spark' runs time series in parallel on a spark cluster in Azure Databricks or Azure Synapse.
<code>inner_parallel</code>	Run components of forecast process inside a specific time series in parallel. Can only be used if <code>parallel_processing</code> is set to NULL or 'spark'.
<code>num_cores</code>	Number of cores to run when parallel processing is set up. Used when running parallel computations on local machine or within Azure. Default of NULL uses total amount of cores on machine minus one. Can't be greater than number of cores on machine minus 1.
<code>seed</code>	Set seed for random number generator. Numeric value.

Details

If individual time series fail during the global or local model update process, they are automatically re-forecast using default local model inputs (the same treatment as new time series). If more than 20\ existing series (with a floor of 10) fail to update, an error is raised directing the user to use `iterate_forecast()` instead.

Reused forecasts are checked against original-scale prepared actuals after refitting and retuning. Quality-rejected series receive one default reforecast through the same path as new series, without LLM quality judgments. Quality-only rejections do not count toward the ordinary execution-failure limit; existing data/provider failures retain that limit. Replacement models are evaluated

using `final_models()` and cannot trigger an unbounded retry. Both new-series defaults and failed-update replacements may select a complete finite least-issues fallback with a warning. Existing rejected defaults are reconsidered from saved usable predictions without fitting. This fallback does not change ordinary update or retuning acceptance. Reused fits do not call `final_models()`; their components must pass hard eligibility and their selected combination must pass applicable quality checks before any reconciliation, including after retuning. Global updates recover each series' saved winning single model or average from existing source forecasts, refit the union of required components, and preserve each selected subset. Requested but unselected models are not added back to the average. All globally selected series must reference one winning global iteration. Mixed global iteration metadata is rejected before refitting rather than split into multiple global updates. Different model subsets within that single iteration and separately selected local winners remain supported. Missing or ambiguous saved winners or selected fits normally require restoring the original artifacts. For older hierarchical global updates whose source files are wholly absent, valid reconciled output and saved global fits permit an explicit compatibility path: reuse each actual fitted model/recipe component, averaging them equally at every source node when there is more than one. This warns once per global group; requested but unfitted models are never invented. Partial source sets, malformed content and storage failures cannot trigger that compatibility path. Earlier Agent versions are not searched to reconstruct lost per-node choices. These are checks on newly generated predictions, not repeated assessments of past iteration winners. A reused hierarchy that is incomplete or contains a rejected node is not reconciled; its covered current series follow the existing default-local forecast path. Existing nodes are matched by identity and surviving bottom membership, not position or node count. Reassigned generated aggregate labels are treated as new nodes rather than inheriting another aggregate's selection. When every predecessor source node has the same selected model or average, new inner hierarchy source nodes inherit that uniform choice. Their component and average forecasts must pass the same checks before publication. This keeps existing global series usable without promoting newly added series to global winners: their ordinary new-series/default-local routing is unchanged. Heterogeneous selections are not extrapolated, removed nodes are not published, and reassigned-label safeguards still apply. An incomplete inner hierarchy continues to use default-local recovery. Stable default run identities and saved forecasts preserve restart safety without an additional acceptance-status column. Unpublished defaults are reassessed from saved predictions before reuse, without fitting. The selected mixture is then reconciled without post-reconciliation future evaluation, whole-set replacement, or late quality-triggered refitting. Legacy second-order differenced original targets without their own starting values require regeneration of prepared data from the original input. On restart, the driver checks fitted objects and required selected forecasts only for series with current best-run metadata. Missing or damaged current results remain eligible for refitting. Every selected forecast must match the expected dates and scenarios from saved preparation, including expanded weekly days. Required preparation and predecessor metadata access failures propagate rather than triggering default fitting. CSV identities remain text. Shared global results are checked and updated as one group, preserving valid local winners. Workers independently check current outputs before refitting and can finish missing best-run logging from complete saved forecasts and existing run-log metrics. Reuse does not repeat quality selection, date conversion, or forecast writes. Every accepted hierarchical global update saves and validates its component and selected-average source files before publishing best-run records. Unselected average files are overwritten with schema-correct empty tables to prevent stale winners after interrupted refits. Restart also checks per-node selections, not just the union of fitted models. Preparation caching and artifact formats are unchanged. Required predecessor and storage access failures remain errors. These checks do not coordinate concurrent writers or guarantee that another attempt cannot overwrite a checked file. Approved custom pools require

explicit allow_iterate_forecast = FALSE, overwrite = TRUE, identical enrollment and approval evidence, cadence, regressors and enabled modes. New observations, positive whole-number horizons, historical windows, cutoffs and backtest settings are allowed in the new parent, subject to approved history and driver requirements. The raw C1 recipe, CSV data on supported storage, RDS objects and no inner parallelism remain required. The completed predecessor, exact selected components, current input signatures and replay controls are pinned before child fitting. Replay retains full custom version IDs and per-series global subsets. Custom logic is never regenerated or tuned; mixed retuning reuses fixed custom results while retuning selected built-ins only. Temporal custom resampling evaluates complete horizons from existing predictors but preserves the original analysis cutoffs and scoring rows. New series and recoverable built-in fitting or forecast-quality failures use the existing default-local fallback with every compatible current-version custom model included first. Defaults also include enrolled foundation models whose credentials are available. Availability is checked once in the coordinator and passed to workers so both validate the same default model pool. Global custom models refit on the compatible current cohort, including new peers, but publish only to surviving series with saved global winners. New peers still receive default-local forecasts, not untested global winners. Changed cohorts never reuse the former fitted state. Source, provider, enrollment, identity and required-artifact failures remain hard errors; custom logic is never repaired or silently dropped. Complete outputs can resume missing completion logging without refits or forecast writes; conflicting or incomplete fitted outputs fail untouched. A completed compatible custom update can be the next update's predecessor. Ordinary EDA post-processing may still call its isolated LLM; approved source and approval payloads are excluded. Replay provenance is not authenticated approval, a sandbox or a concurrent-writer transaction. A built-in-only parent cannot adopt a custom predecessor without explicit enrollment.

Value

Nothing

Examples

```
## Not run:
# load example data
hist_data <- timetk::m4_monthly %>%
  dplyr::filter(date >= "2013-01-01") %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id))

# set up Finn project
project <- set_project_info(
  project_name = "Demo_Project",
  combo_variables = c("id"),
  target_variable = "value",
  date_type = "month"
)

# set up LLM
llm <- ellmer::chat_azure_openai(model = "gpt-4o-mini")

# set up agent info
agent_info <- set_agent_info(
  project_info = project,
```

```

    llm = llm,
    input_data = hist_data,
    forecast_horizon = 6,
    hist_end_date = as.Date("2014-12-01")
  )

  # run the forecast iteration process
  iterate_forecast(
    agent_info = agent_info,
    max_iter = 3,
    weighted_mape_goal = 0.03
  )

  # update the forecast with latest data and inputs
  agent_info <- set_agent_info(
    project_info = project,
    llm = llm,
    input_data = hist_data,
    forecast_horizon = 6,
    hist_end_date = as.Date("2014-12-01"),
    overwrite = TRUE # required to update the agent for latest data and inputs
  )

  update_forecast(
    agent_info = agent_info,
    weighted_mape_goal = 0.03
  )

  ## End(Not run)

```

validate_prepared_input

Validate Prepared Input Data

Description

Checks a prepared input table against the rules `prep_data()` enforces and reports how preparation will treat the data, without starting a run or writing anything to disk.

Usage

```

validate_prepared_input(
  input_data,
  combo_variables,
  target_variable,
  date_type,
  forecast_horizon,
  external_regressors = NULL,
  hist_start_date = NULL,

```

```

    hist_end_date = NULL,
    combo_cleanup_date = NULL,
    fiscal_year_start = 1,
    clean_missing_values = TRUE,
    forecast_approach = "bottoms_up",
    parallel_processing = NULL,
    back_test_scenarios = NULL,
    back_test_spacing = NULL
  )

```

Arguments

<code>input_data</code>	A standard data frame, tibble, or spark data frame using sparklyr of historical time series data. Can also include external regressors for both historical and future data. Leading and trailing whitespace in character combo-variable values is removed before Finn creates internal series identifiers.
<code>combo_variables</code>	List of column headers within input data to be used to separate individual time series.
<code>target_variable</code>	The column header formatted as a character value within input data you want to forecast.
<code>date_type</code>	The date granularity of the input data. Finn accepts the following as a character string: day, week, month, quarter, year.
<code>forecast_horizon</code>	Number of periods to forecast into the future.
<code>external_regressors</code>	List of column headers within input data to be used as features in multivariate models. Controls built-in R1/R2 recipes only. C1 retains eligible original numeric, character, factor, logical and Date columns independently of this list in bottoms_up runs; hierarchical C1 nodes keep only these aggregated regressors. Each custom workflow uses only its declared predictors plus Date and Combo.
<code>hist_start_date</code>	Date value of when your input_data starts. Default of NULL uses earliest date value in input_data.
<code>hist_end_date</code>	Date value of when your input_data ends. Default of NULL uses the latest date value in input_data.
<code>combo_cleanup_date</code>	Date value to remove individual time series that don't contain non-zero values after that specified date. Default of NULL is to not remove any time series and attempt to forecast all time series.
<code>fiscal_year_start</code>	Month number of start of fiscal year of input data, aids in building out date features. Formatted as a numeric value. Default of 1 assumes fiscal year starts in January.

clean_missing_values

If TRUE, cleans missing values. Only impute values for missing data within an existing series, and does not add new values onto the beginning or end, but does provide a value of 0 for said values.

forecast_approach

Forecast approach, as passed to `prep_data()`. It is validated but does not change the checks: every approach expects bottom-level series, so totals mixed with their parts fail P7 for all.

parallel_processing

Default of NULL runs no parallel processing and forecasts each individual time series one after another. Value of 'local_machine' leverages all cores on current machine Finn is running on. Value of 'spark' runs time series in parallel on a spark cluster in Azure Databricks/Synapse.

back_test_scenarios

Number of back-test scenarios, as passed to `prep_models()`. Default of NULL chooses them from the history length.

back_test_spacing

Periods between back-test scenarios, as passed to `prep_models()`. Default of NULL uses the date type default.

Details

Input rules R1-R8, R10 and R11 reuse the checks in `prep_data()`, so a failure here carries the same message `prep_data()` would raise. R9 is stricter than `prep_data()`: it also checks that `date_type` matches the spacing of the dates. Preparation checks (P1-P8) describe what `prep_data()` and the train/test split planning will do with the data, such as future rows, padded gaps, missing values, series removed by `combo_cleanup_date` and short histories. A check fails only when the data cannot be prepared correctly; behavior that is applied automatically is returned with status "report".

Rules:

- R1 combo variables, the target and external regressors exist.
- R2 Date is not a combo variable, the target or an external regressor.
- R3 the date column is named Date and has class Date.
- R4 combo variables are not date columns.
- R5 the target is numeric.
- R6 combo values are not blank after trimming whitespace.
- R7 each combo and Date pair appears once.
- R8 month, quarter and year dates fall on one day of the month.
- R9 `date_type` is valid and is not finer or coarser than the data.
- R10 `fiscal_year_start` is a number from 1 to 12.
- R11 Spark tables run with Spark parallel processing.
- P1 history end date and the rows treated as future periods.
- P2 external regressors with future values and their coverage.
- P3 missing periods that preparation pads.

- P4 missing, NaN and infinite target values.
- P5 weekly dates share a weekday; quarterly and yearly dates share a period month.
- P6 series removed by combo_cleanup_date.
- P7 total or subtotal series mixed with the series they sum.
- P8 history supports the forecast horizon and back-test splits.

Preparation checks run only when every input rule passes, and a check that depends on a failed check is skipped. Spark tables are collected locally after R11 is checked.

Value

A list with:

- passed: TRUE when no check failed.
- checks: tibble with one row per rule and columns Rule, Description, Status ("pass", "fail", "report" or "skipped") and Message.
- summary: list with the history start and end dates, row, series, history and future row counts, date_type and forecast_horizon.
- details: list of tibbles and vectors behind the preparation checks: future_regressors, future_regressor_coverage, gaps, target_values, cleanup_combos, total_like_values, aggregate_levels, splits and short_history_combos.

Examples

```
data_tbl <- timetk::m4_monthly %>%
  dplyr::rename(Date = date) %>%
  dplyr::mutate(id = as.character(id)) %>%
  dplyr::filter(Date >= "2010-01-01")

result <- validate_prepared_input(
  input_data = data_tbl,
  combo_variables = "id",
  target_variable = "value",
  date_type = "month",
  forecast_horizon = 3
)

result$passed
result$checks
```

Index

ask_agent, 3

chat_copilot, 4
chat_copilot(), 6, 34, 43
create_custom_model, 5
create_custom_model(), 4, 9–12, 22, 41, 42, 44
custom_model_control, 9
custom_model_control(), 5, 6, 8
custom_model_examples_template, 10
custom_model_examples_template(), 5, 6, 8
custom_model_feedback, 11
custom_model_feedback(), 6, 8, 42

detect_hierarchy, 12

ensemble_models, 13

final_models, 15
final_models(), 22, 34, 50, 52
forecast_time_series, 18

get_agent_forecast, 23
get_agent_forecast(), 27
get_best_agent_run, 24
get_eda_data, 26
get_forecast_data, 27
get_forecast_data(), 22, 23
get_prepped_data, 28
get_prepped_models, 29
get_run_info, 30
get_summarized_models, 31
get_trained_models, 32

iterate_forecast, 33
iterate_forecast(), 23, 27

list_models, 36

prep_data, 36
prep_data(), 12, 13, 41, 54, 56
prep_models, 39
prep_models(), 6, 8, 22, 49, 56
print.finnts_custom_model, 42

set_agent_info, 43
set_agent_info(), 35
set_project_info, 45
set_run_info, 47
set_run_info(), 14, 15, 19, 27–29, 33, 37, 39, 40, 48

train_models, 48

update_forecast, 50
update_forecast(), 23, 35, 44

validate_prepared_input, 54