Package 'quallmer'

October 21, 2026

Type Package

Title Qualitative Analysis with Large Language Models

Version 0.6.0

Description Tools for AI-assisted qualitative data coding using large language models ('LLMs') via the 'ellmer' package, supporting providers including 'OpenAI', 'Anthropic', 'Google', 'Azure', and local models via 'Ollama'. Provides a 'codebook'-based workflow for defining coding instructions and applying them to texts, images, audio recordings, and other data, with optional transcription of audio and translation of texts before coding through 'DeepL', 'Google', 'Qwen-MT', or a chat model. Includes built-in 'codebooks' for common applications such as sentiment analysis and policy coding, and functions for creating custom 'codebooks' for specific research questions. Supports systematic replication across models and settings, computing inter-coder reliability statistics including Krippendorff's alpha (Krippendorff 2019, <doi:10.4135/9781071878781>) and Fleiss' kappa (Fleiss 1971, <doi:10.1037/h0031619>), as well as gold-standard validation metrics including accuracy, precision, recall, and F1 scores following Sokolova and Lapalme (2009, <doi:10.1016/j.ipm.2009.03.002>). Provides audit trail functionality for documenting coding workflows following Lincoln and Guba's (1985, ISBN:0803924313) framework for establishing trustworthiness in qualitative research.

License GPL (>= 3)

URL https://quallmer.github.io/quallmer/

Depends R (>= 3.5.0), ellmer (>= 0.5.0)

Imports cli, curl, digest, httr2, jsonlite, lifecycle, rlang, stats, tibble, vctrs, xml2

Encoding UTF-8

LazyData true

Suggests av, ggplot2, janitor, keyring, knitr, magick, rmarkdown, testthat (>= 3.0.0), kableExtra, mockery, quanteda, quanteda.tidy, withr, yardstick

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Seraphine F. Maerz [aut, cre] (ORCID:
<https://orcid.org/0000-0002-7173-9617>),
Kenneth Benoit [aut] (ORCID: <https://orcid.org/0000-0002-0797-564X>)

Maintainer Seraphine F. Maerz <seraphine.maerz@unimelb.edu.au>

Repository CRAN

Date/Publication 2026-10-21 03:30:14 UTC

Contents

accessors . . . 2
as_qlm_coded . . . 4
corpus.qlm_coded . . . 8
data_codebook_immigration . . . 9
data_codebook_sentiment . . . 11
data_corpus_LMRDsample . . . 12
data_corpus_manifsentsUK2010sample . . . 13
data_corpus_MPexamples . . . 14
data_corpus_ms2020sample . . . 16
qlm_backfill . . . 17
qlm_code . . . 19
qlm_codebook . . . 29
qlm_compare . . . 31
qlm_failures . . . 36
qlm_meta . . . 37
qlm_register_provider . . . 39
qlm_replicate . . . 40
qlm_segment . . . 42
qlm_trail . . . 45
qlm_transcribe . . . 47
qlm_translate . . . 50
qlm_validate . . . 55

Index **59**

`accessors` *Accessor functions for quallmer objects*

Description

Functions to safely access and modify metadata from quallmer objects (`qlm_coded`, `qlm_comparison`, `qlm_validation`, `qlm_codebook`). These functions provide a stable API for accessing object metadata without directly manipulating internal attributes.

Metadata types

quallmer objects store metadata in three categories:

User metadata (`type = "user"`):

- `name`: Run identifier (settable)
- `notes`: Descriptive notes (settable)
- Plus any custom fields added via `as_qlm_coded(..., metadata = list(...))`

Object metadata (`type = "object"`):

- `call`: Function call that created the object
- `parent`: Parent run name (for replications)
- `batch`: Whether batch processing was used
- `chat_args`: Arguments passed to the LLM chat
- `execution_args`: Arguments for parallel/batch execution
- `n_units`: Number of coded units
- `input_type`: Type of input ("text", "image", "audio", or "human")
- `source`: Coding source ("human" or "llm")
- `is_gold`: Whether this is a gold standard

System metadata (`type = "system"`):

- `timestamp`: When the object was created
- `ellmer_version`: Version of ellmer package
- `quallmer_version`: Version of quallmer package
- `R_version`: Version of R

Functions

- `qlm_meta()`: Get metadata fields
- `qlm_meta<-()`: Set user metadata fields (only `name` and `notes`)
- `codebook()`: Extract codebook from coded objects
- `inputs()`: Extract original input data
- `qlm_failures()`: List the units a coding run failed on, with reasons

See Also

- `qlm_code()` for creating coded objects
- `as_qlm_coded()` for converting human-coded data
- `qlm_trail()` for viewing coding history

Examples

```
## Not run:
# Create a coded object
texts <- c("I love this!", "Terrible.", "It's okay.")
coded <- qlm_code(
  texts,
  data_codebook_sentiment,
  model = "openai/gpt-4o-mini",
  name = "run1",
  notes = "Initial coding run"
)

# Access metadata
qlm_meta(coded, "name")              # Get run name
qlm_meta(coded, type = "user")       # Get all user metadata
qlm_meta(coded, type = "system")     # Get system metadata

# Modify user metadata
qlm_meta(coded, "name") <- "updated_run1"
qlm_meta(coded, "notes") <- "Revised notes"

# Extract components
codebook(coded)                      # Get the codebook
inputs(coded)                        # Get original texts

# Custom metadata from human coding
human_data <- data.frame(
  .id = 1:5,
  sentiment = c("pos", "neg", "pos", "neg", "pos")
)
human_coded <- as_qlm_coded(
  human_data,
  name = "coder_A",
  metadata = list(
    coder_name = "Dr. Smith",
    experience = "5 years"
  )
)

# Access custom metadata
qlm_meta(human_coded, "coder_name")  # "Dr. Smith"
qlm_meta(human_coded, type = "user") # All user fields

## End(Not run)
```

---

as_qlm_coded *Convert coded data to qlm_coded format*

Description

Converts a data frame or quanteda corpus of coded data (human-coded or from external sources) into a `qlm_coded` object. This enables provenance tracking and integration with `qlm_compare()`, `qlm_validate()`, and `qlm_trail()` for coded data alongside LLM-coded results.

Usage

```
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list(),
  qlm_segment = FALSE,
  source_text = NULL
)

## S3 method for class 'data.frame'
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list(),
  qlm_segment = FALSE,
  source_text = NULL
)

## Default S3 method:
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list(),
  qlm_segment = FALSE,
  source_text = NULL
)
```

Arguments

x	A data frame or quanteda corpus object containing coded data. For data frames: Must include a column with unit identifiers (default ".id"). For corpus objects: Document variables (docvars) are treated as coded variables, and document names are used as identifiers by default.
id	For data frames: Name of the column containing unit identifiers (supports both quoted and unquoted). Default is NULL, which looks for a column named ".id". Can be an unquoted column name (id = doc_id) or a quoted string (id = "doc_id"). For corpus objects: NULL (default) uses document names from names(x), or specify a docvar name (quoted or unquoted) to use as identifiers.
name	Character. a string identifying this coding run (e.g., "Coder_A", "expert_rater", "Gold_Standard"). Default is NULL.
is_gold	Logical. If TRUE, marks this object as a gold standard for automatic detection by qlm_validate(). When a gold standard object is passed to qlm_validate(), the gold = parameter becomes optional. Default is FALSE.
codebook	Optional list containing coding instructions. Can include: name Name of the coding scheme instructions Text describing coding instructions schema NULL (not used for human coding) If NULL (default), a minimal placeholder codebook is created. Passing the qlm_codebook() the LLM coded against gives the human data the same measurement levels, and a column that is a type_enum() declared "ordinal" there is stored as an ordered factor in the enum's order, so qlm_compare() and qlm_validate() rank it as they rank the LLM's; a value outside the enum is an error. Without a codebook, order a text scale yourself with factor(x, levels = c(...), ordered = TRUE). The schema itself is not kept.
texts	Optional vector of original texts or data that were coded. Should correspond to the .id values in data. If provided, enables more complete provenance tracking.
notes	Optional character string with descriptive notes about this coding. Useful for documenting details when viewing results in qlm_trail(). Default is NULL.
metadata	Optional list of metadata about the coding process. Can include any relevant information such as: coder_name Name of the human coder coder_id Identifier for the coder training Description of coder training date Date of coding The function automatically adds timestamp, n_units, notes, and source = "human".
qlm_segment	Logical. If TRUE, converts the data to a segmented quanteda corpus suitable for unitizing comparison with qlm_compare(). Requires a text column in x and a source_text argument. Default is FALSE.
source_text	A named character vector of source texts. Required when qlm_segment = TRUE. Names must match docid values in x (or a single unnamed string for single-document data). Used to compute character-level segment positions for unitizing reliability.

Details

When printed, objects created with `as_qlm_coded()` display "Source: Human coder" instead of model information, clearly distinguishing human from LLM coding.

Gold Standards:

Objects marked with `is_gold = TRUE` are automatically detected by `qlm_validate()`, allowing simpler syntax:

```
# With is_gold = TRUE
gold <- as_qlm_coded(gold_data, name = "Expert", is_gold = TRUE)
qlm_validate(coded1, coded2, gold, by = "sentiment")  # gold = not needed!

# Without is_gold (or explicit gold =)
gold <- as_qlm_coded(gold_data, name = "Expert")
qlm_validate(coded1, coded2, gold = gold, by = "sentiment")
```

Value

A `qlm_coded` object (tibble with additional class and attributes) for provenance tracking. When `is_gold = TRUE`, the object is marked as a gold standard in its attributes.

See Also

`qlm_code()` for LLM coding, `qlm_compare()` for inter-rater reliability, `qlm_validate()` for validation against gold standards, `qlm_trail()` for provenance tracking.

Examples

```
# Basic usage with data frame (default .id column)
human_data <- data.frame(
  .id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)

coder_a <- as_qlm_coded(human_data, name = "Coder_A")
coder_a

# Use custom id column with NSE (unquoted)
data_with_custom_id <- data.frame(
  doc_id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)
coder_custom <- as_qlm_coded(data_with_custom_id, id = doc_id, name = "Coder_C")

# Or use quoted string
coder_custom2 <- as_qlm_coded(data_with_custom_id, id = "doc_id", name = "Coder_D")

# Create a gold standard from data frame
gold <- as_qlm_coded(
  human_data,
  name = "Expert",
```

```
  is_gold = TRUE
)

# Validate with automatic gold detection
coder_b_data <- data.frame(
  .id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)
coder_b <- as_qlm_coded(coder_b_data, name = "Coder_B")

# No need for gold = when gold object is marked (NSE works for 'by' too)
qlm_validate(coder_a, coder_b, gold = gold, by = sentiment, level = "nominal")

# Create from corpus object (simplified workflow)
data("data_corpus_manifsentsUK2010sample")
crowd <- as_qlm_coded(
  data_corpus_manifsentsUK2010sample,
  is_gold = TRUE
)
# Document names automatically become .id, all docvars included

# Use a docvar as identifier, with NSE (unquoted) or a quoted string. It
# must identify each unit uniquely: a party repeats across sentences, so
# give the corpus a sentence identifier first
if (requireNamespace("quanteda", quietly = TRUE)) {
  corp <- data_corpus_manifsentsUK2010sample
  quanteda::docvars(corp, "sentence_id") <- seq_len(quanteda::ndoc(corp))
  crowd_sent <- as_qlm_coded(corp, id = sentence_id, is_gold = TRUE)
  crowd_sent2 <- as_qlm_coded(corp, id = "sentence_id", is_gold = TRUE)
}

# With complete metadata
expert <- as_qlm_coded(
  human_data,
  name = "expert_rater",
  is_gold = TRUE,
  codebook = list(
    name = "Sentiment Analysis",
    instructions = "Code overall sentiment as positive or negative"
  ),
  metadata = list(
    coder_name = "Dr. Smith",
    coder_id = "EXP001",
    training = "5 years experience",
    date = "2024-01-15"
  )
)
```

`corpus.qlm_coded` *Convert translations and coded text fields to a corpus*

Description

Converts a qlm_translate() result, or a text column of a qlm_coded object, to a quanteda corpus. Quanteda is optional and is needed only when calling its corpus() generic.

Usage

```
corpus.qlm_coded(x, text_field = NULL, ...)

corpus.qlm_translation(x, ...)
```

Arguments

x	A qlm_translation or a qlm_coded object.
text_field	For a qlm_coded object, the name of the character column to use as corpus texts. It can be omitted only when there is exactly one character output column.
...	Reserved; must be empty.

Value

A quanteda corpus with the translation names, or .id, as document names. If the texts came from a corpus, its docvars and system and user metadata are restored, and text summary caches are cleared. Translation provenance is stored in meta$user$quallmer$translation. For a coded object, the other output columns, coding metadata and codebook are stored in meta$user$quallmer$coding.

See Also

qlm_translate(), as_qlm_coded()

Examples

```
## Not run:
translated <- qlm_translate(c(a = "Bonjour"), "deepl/quality_optimized")
quanteda::corpus(translated)

summaries <- qlm_code(translated, my_summary_codebook, model = "openai/gpt-4o-mini")
quanteda::corpus(summaries, text_field = "summary")

## End(Not run)
```

---

data_codebook_immigration

Immigration policy codebook based on Benoit et al. (2016)

Description

A qlm_codebook object defining instructions for annotating whether a text pertains to immigration policy and, if so, the stance toward immigration openness. This codebook replicates the crowd-sourced annotation task from Benoit et al. (2016) and is designed to work with data_corpus_manifsentsUK2010sample.

Usage

```
data_codebook_immigration
```

Format

A `qlm_codebook` object containing:

name Task name: "Immigration policy coding from Benoit et al. (2016)"

instructions Coding instructions for identifying whether sentences from UK 2010 election manifestos pertain to immigration policy, and if so, rating the policy position expressed

schema Response schema with two fields: `llm_immigration_label` (Enum: "Not immigration" or "Immigration" indicating whether the sentence relates to immigration policy), and `llm_immigration_position` (Integer from -1 to 1, where -1 = pro-immigration, 0 = neutral, and 1 = anti-immigration)

input_type "text"

levels Named character vector: llm_immigration_label = "nominal", llm_immigration_position = "ordinal"

References

Benoit, K., Conway, D., Lauderdale, B.E., Laver, M., & Mikhaylov, S. (2016). Crowd-sourced Text Analysis: Reproducible and Agile Production of Political Data. *American Political Science Review*, 110(2), 278–295. doi:10.1017/S0003055416000058

See Also

`qlm_codebook()`, `qlm_code()`, data_corpus_manifsentsUK2010sample

Examples

```
# View the codebook
data_codebook_immigration

## Not run:
# Use with UK manifesto sentences (requires API key)
if (requireNamespace("quanteda", quietly = TRUE)) {
  coded <- qlm_code(data_corpus_manifsentsUK2010sample,
                    data_codebook_immigration,
                    model = "openai/gpt-4o-mini")

  # Compare with crowd-sourced annotations
  crowd <- as_qlm_coded(
    data.frame(
      .id = docnames(data_corpus_manifsentsUK2010sample),
      docvars(data_corpus_manifsentsUK2010sample)
    ),
    is_gold = TRUE
  )

  qlm_validate(coded, gold = crowd)
```

```
}

## End(Not run)
```

`data_codebook_sentiment` *Sentiment analysis codebook for movie reviews*

Description

A `qlm_codebook` object defining instructions for sentiment analysis of movie reviews. Designed to work with data_corpus_LMRDsample but with an expanded polarity scale that includes a "mixed" category.

Usage

```
data_codebook_sentiment
```

Format

A `qlm_codebook` object containing:

name Task name: "Movie Review Sentiment"

instructions Coding instructions for analyzing movie review sentiment

schema Response schema with two fields: `polarity` (Enum of "neg", "mixed", or "pos") and `rating` (Integer from 1 to 10)

role Expert film critic persona

input_type "text"

See Also

`qlm_codebook()`, `qlm_code()`, `qlm_compare()`, data_corpus_LMRDsample

Examples

```
# View the codebook
data_codebook_sentiment

## Not run:
# Use with movie review corpus (requires API key)
coded <- qlm_code(data_corpus_LMRDsample[1:10],
                  data_codebook_sentiment,
                  model = "openai")

# Create multiple coded versions for comparison
coded1 <- qlm_code(data_corpus_LMRDsample[1:20],
                   data_codebook_sentiment,
                   model = "openai/gpt-4o-mini")
```

```
coded2 <- qlm_code(data_corpus_LMRDsample[1:20],
                   data_codebook_sentiment,
                   model = "openai/gpt-4o")

# Compare inter-rater reliability
comparison <- qlm_compare(coded1, coded2, by = "rating", level = "interval")
print(comparison)

## End(Not run)
```

`data_corpus_LMRDsample` *Sample from Large Movie Review Dataset (Maas et al. 2011)*

Description

A sample of 100 positive and 100 negative reviews from the Maas et al. (2011) dataset for sentiment classification. The original dataset contains 50,000 highly polar movie reviews.

Usage

```
data_corpus_LMRDsample
```

Format

The corpus docvars consist of:

docnumber serial (within set and polarity) document number

rating user-assigned movie rating on a 1-10 point integer scale

polarity either `neg` or `pos` to indicate whether the movie review was negative or positive. See Maas et al (2011) for the cut-off values that governed this assignment.

Source

http://ai.stanford.edu/~amaas/data/sentiment/

References

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). "Learning Word Vectors for Sentiment Analysis". The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

See Also

data_codebook_sentiment for an example codebook and usage with this corpus

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_LMRDsample)

  # Sample a few reviews
  head(data_corpus_LMRDsample, 3)
}
```

`data_corpus_manifsentsUK2010sample` *Sample of UK manifesto sentences 2010 crowd-annotated for immigration*

Description

A corpus of sentences sampled from from publicly available party manifestos from the United Kingdom from the 2010 election. Each sentence has been rated in terms of its classification as pertaining to immigration or not and then on a scale of favorability or not toward open immigration policy (as the mean score of crowd coders on a scale of -1 (favours open immigration policy), 0 (neutral), or 1 (anti-immigration).

The sentences were sampled from the corpus used in Benoit et al. (2016) doi:10.1017/S0003055416000058, which contains more information on the crowd-sourced annotation approach.

Usage

```
data_corpus_manifsentsUK2010sample
```

Format

A corpus object. The corpus consists of 155 sentences randomly sampled from the party manifestos, with an attempt to balance the sentencs according to their categorisation as pertaining to immigration or not, as well as by party. The corpus contains the following document-level variables:

party factor; abbreviation of the party that wrote the manifesto.

partyname factor; party that wrote the manifesto.

year integer; 4-digit year of the election.

immigration_label Factor indicating whether the majority of crowd workers labelled a sentence as referring to immigration or not. The variable has missing values (NA) for all non-annotated manifestos.

immigration_mean numeric; the direction of statements coded as "Immigration" based on the aggregated crowd codings. The variable is the mean of the scores assigned by workers who coded a sentence and who allocated the sentence to the "Immigration" category. The variable ranges from -1 (Favorable and open immigration policy) to +1 ("Negative and closed immigration policy").

immigration_n integer; the number of coders who contributed to the mean score `immigration_mean`.

immigration_position integer; a thresholded version of `immigration_mean` coded as -1 (pro-immigration, mean < -0.5), 0 (neutral, -0.5 <= mean <= 0.5), or 1 (anti-immigration, mean > 0.5). Set to `NA` for non-immigration sentences.

References

Benoit, K., Conway, D., Lauderdale, B.E., Laver, M., & Mikhaylov, S. (2016). Crowd-sourced Text Analysis: Reproducible and Agile Production of Political Data. *American Political Science Review*, 100,(2), 278–295. doi:10.1017/S0003055416000058

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_manifsentsUK2010sample)
}
```

---

data_corpus_MPexamples

Manifesto Project example manifestos and gold-standard segmentation

---

Description

Two datasets derived from Appendix 2 of Klingemann et al. (2006), which provides worked examples of the Manifesto Project quasi-sentence coding scheme.

`data_corpus_MPexamples` is a two-document corpus containing the full source texts of the Liberal-SDP Alliance 1983 UK election manifesto and the New Zealand National Party 1972 election manifesto, reconstructed by joining the quasi-sentences from the gold-standard annotation.

`data_corpus_MPexamplesseg` is the corresponding gold-standard segmented corpus, produced by converting the Manifesto Project's human-coded quasi-sentences via `as_qlm_coded()` with `qlm_segment = TRUE`. It is marked as a gold standard (`is_gold = TRUE`) and can be passed directly to `qlm_compare()` alongside output from `qlm_segment()` to compute Krippendorff's alpha for unitizing.

Usage

```
data_corpus_MPexamples

data_corpus_MPexamplesseg
```

Format

`data_corpus_MPexamples`: A corpus with 2 documents and the following document-level variables:

country Character. Country of origin: `"UK"` or `"NZ"`.

party Character. Party name: "Liberal-SDP Alliance" or "National Party".

year Integer. Election year: 1983 or 1972.

data_corpus_MPexamplesseg: A segmented corpus with 178 quasi-sentences (107 Liberal-SDP, 71 NZ National Party) and the following document-level variables:

docid Character. Source document identifier ("Liberal_SDP_1983" or "NZ_NP_1972").

segid Integer. Quasi-sentence index within the source document.

char_start Integer. Start character position in the source text.

char_end Integer. End character position in the source text.

manifesto Character. Manifesto Project manifesto label ("Liberal-SDP 1983" or "NP 1972").

country Character. Country of origin: "UK" or "NZ".

per Integer. Manifesto Project policy category code.

An object of class corpus (inherits from character) of length 178.

References

Klingemann, H. D., Volkens, A., Bara, J., Budge, I., & McDonald, M. D. (2006). *Mapping Policy Preferences II: Estimates for Parties, Electors, and Governments in Eastern Europe, European Union, and OECD 1990–2003*. Oxford University Press.

See Also

qlm_segment(), as_qlm_coded(), qlm_compare()

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the source texts
  summary(data_corpus_MPexamples)

  # Subset to one manifesto
  quanteda::corpus_subset(data_corpus_MPexamples, country == "NZ")

  # Gold-standard segmentation for the NZ manifesto
  quanteda::corpus_subset(data_corpus_MPexamplesseg,
                          quanteda::docvars(data_corpus_MPexamplesseg,
                                            "docid") == "NZ_NP_1972")
}
```

`data_corpus_ms2020sample` *Sample corpus of political speeches from Maerz & Schneider (2020)*

Description

A corpus of 100 speeches from the Maerz & Schneider (2020) corpus, balanced across regime types (50 autocracies, 50 democracies). This sample is included in the package for demos and testing. The full corpus of 4,740 speeches is available in the package's pkgdown examples folder.

Usage

```
data_corpus_ms2020sample
```

Format

A corpus object. The corpus consists of 100 speeches randomly sampled from 40 heads of government across 27 countries, balanced by regime type. The corpus contains the following document-level variables:

speaker Character. Name of the head of government.

country Character. Country name.

regime Factor. Regime type: "Democracy" or "Autocracy".

score Numeric. Original dictionary-based liberal-illiberal score.

date Date. Date of the speech.

title Character. Title of the speech.

References

Maerz, S. F., & Schneider, C. Q. (2020). Comparing public communication in democracies and autocracies: Automated text analyses of speeches by heads of government. *Quality & Quantity*, 54, 517-545. doi:10.1007/s11135019008857

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_ms2020sample, n = 10)

  # Regime distribution
  table(data_corpus_ms2020sample$regime)

  # View a sample speech
  cat(data_corpus_ms2020sample[1])
}
```

qlm_backfill *Re-code the units a run failed on*

Description

A coding run over a real corpus rarely comes back complete. Requests time out or are rate-limited, a provider refuses a text on one pass and codes it on the next, an endpoint accepts a schema and ignores it for a few units. The failed units sit in the qlm_coded object as NA rows, listed by qlm_failures(). qlm_backfill() re-codes only those units and merges what comes back into the original object. Everything that succeeded the first time is left exactly as it was.

Usage

```
qlm_backfill(x, ..., model = NULL, passes = 2L)
```

Arguments

x	qlm_coded; a coded object produced by qlm_code() or qlm_replicate().
...	optional overrides passed to qlm_code() for the backfill passes, such as params, max_active or on_error. Any setting not overridden is restored from the original run, as qlm_replicate() does, with the same rule for credentials and endpoint settings when the provider changes. The codebook, batch, name and backfill cannot be set: passes is the only bound on the number of passes. Nor can include_tokens and include_cost: usage is recorded as the run recorded it, so that the merged columns mean one thing. prices may be given, to cost the passes where the run's own rates do not carry (a batch-coded run's rates do not cover its parallel passes), and needs the run to have recorded token counts and cost of its own to add to.
model	character or NULL; the model for the passes, in the form used by qlm_code(). NULL (default) uses the run's own model.
passes	A single positive integer giving the maximum number of backfill passes. Default is 2. This counts total passes, not additional retries. Backfilling stops early when a pass recovers no units, since the failures that remain are then evidently not transient.

Details

By default the passes use the run's own model, codebook and settings, so the result is what the run should have produced. A different model can be given, for units the original model consistently refuses or cannot fit in its context window; the object then records which units were coded by which model, and print() and qlm_trail() say so, since a result coded by two instruments has to be disclosed as one.

Which units are re-coded is decided afresh on every pass, from the object as it then stands, by the same test qlm_failures() uses: a unit that carries an .error, or whose required scalar properties are all NA. Two kinds of failure are left alone, because re-sending the same request cannot change the outcome:

- a text the provider rejected as longer than the model's context window, unless the model or endpoint changes;
- a response cut off at the `max_tokens` limit, unless the model or endpoint changes or the backfill raises the limit, by passing `params(max_tokens = )` higher than the run's own. Other `params` leave the limit where it was, and so leave those units alone.

Content refusals are deliberately retried. They look deterministic and are not: the same document is refused on one pass and coded on the next, at more than one provider.

Each pass is an ordinary `qlm_code()` call over the failed units, on the path the original run took (a run that fell back to JSON mode is backfilled in JSON mode; with a different endpoint the path is chosen afresh), and always as a parallel call: a run coded through the batch API is backfilled through the parallel API, with the same model and settings, and any batch-only arguments (`path`, `wait`, `ignore_hash`) set aside. A pass that fails outright on the first attempt is an error, since nothing has been gained yet and the cause is most likely configuration; on a later pass it is a warning, and what earlier passes recovered is kept. The failed pass is still recorded, with the units it attempted, no recoveries and the error, since the provider may have billed it, and for the same reason the token and cost columns of the units it attempted become `NA`: how much was billed is not known, so no total for them is.

Units are identified by `.id` throughout: the failed units' inputs are looked up by `.id`, so an object whose rows have been reordered or subset is backfilled correctly, and the merge is by `.id`. Rows keep their order; a unit is replaced only when the retry produced a usable coding, so a retry that failed again never overwrites anything, though its `.error` is recorded as the latest reason. Token and cost columns, when present, are summed across all attempts, since a failed request may still have been billed; a total is `NA` when any attempt's figure is, because `NA` means the provider did not report it, not that nothing was billed, so a retry's known figure cannot stand in for the whole. The passes are recorded in the object metadata as `backfill`, one entry per pass with its timestamp, the model if it or the endpoint differed from the run's, the overrides, the `.id`s attempted and recovered, where its cost came from when that was not where the run's did, and for a pass that failed outright its error, so the result can say which of its rows came from which pass and which model. A pass whose cost came from somewhere else than the run's, other supplied rates, ellmer's own table where the run rested on supplied rates, or nowhere where the run was priced, is disclosed by `print()` and `qlm_trail()` beside the run's own cost note, since part of the cost column then rests on it. `qlm_trail()` redacts any credential among a pass's overrides as it does the run's own, and a pass replayed from a trail does not send a redacted value. `qlm_replicate()` replays these passes on a replication, so that a replication of a completed run is completed on the same terms.

Value

`x`, with the recovered units filled in and `backfill` added to its object metadata. The run name, parent, codebook and inputs are unchanged.

See Also

`qlm_failures()` for the units a run failed on and why; `qlm_code()`, whose `backfill` completes a run in the same call; `qlm_replicate()` to re-run a whole coding.

Examples

```
# A run that came back incomplete, and what qlm_backfill() made of it. Both
```

```
# were coded once and saved with the package (see data_creation/ in the
# source), so they can be looked at without a key.
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))
incomplete <- examples$example_coded_incomplete
incomplete
qlm_failures(incomplete)

# What qlm_backfill(incomplete) returned: the timed-out unit re-coded, the
# responses cut off at max_tokens left alone, and the pass on record
filled <- examples$example_coded_backfilled
filled
qlm_failures(filled)
qlm_meta(filled, "backfill", type = "object")

## Not run:
filled <- qlm_backfill(incomplete)

# Responses cut off at the output limit are retried only with a higher one
filled <- qlm_backfill(filled, params = ellmer::params(max_tokens = 2000))

# Units one model refuses or cannot fit, coded by another; the result
# records which units came from which model
filled <- qlm_backfill(filled, model = "deepseek/deepseek-chat")

## End(Not run)
```

---

qlm_code *Code qualitative data with an LLM*

Description

Applies a codebook to input data using a large language model, returning a rich object that includes the codebook, execution settings, results, and metadata for reproducibility.

Usage

```
qlm_code(
  x,
  codebook,
  model,
  ...,
  batch = FALSE,
  tools = NULL,
  structured = c("auto", "structured", "json"),
  json_retries = 2L,
  on_error = c("continue", "return", "stop"),
  backfill = FALSE,
  prices = NULL,
```

```
  name = NULL,
  notes = NULL
)
```

Arguments

x	character; the input data: texts for a text codebook, file paths or URLs for an image codebook (see the section on image input), or file paths for an audio codebook (see the section on audio input), or file paths, YouTube links and URLs of video files for a video codebook (see the section on video input). Named vectors will use names as identifiers in the output; unnamed vectors will use sequential integers. The identifiers become the `.id` column, on which every later operation keys, so names must be unique.
codebook	qlm_codebook; a codebook created with `qlm_codebook()`. Also accepts deprecated `task()` objects for backward compatibility.
model	character; the provider (and optionally model) name in the form `"provider/model"` or `"provider"` (which will use the default model for that provider). Native prefixes are passed to `ellmer::chat()`. Registered prefixes, such as `"dashscope/qwen-plus"`, resolve through `qlm_register_provider()` and require an explicit model name. Examples: `"openai/gpt-4o-mini"`, `"anthropic/claude-3-5-sonnet-20241022"`, `"ollama/llama3.2"`, `"openai"` (uses default OpenAI model).
...	Additional arguments passed to `ellmer::chat()`, `ellmer::parallel_chat_structured()`, or `ellmer::batch_chat_structured()`. Arguments recognized by `ellmer::parallel_chat_structu` or `ellmer::batch_chat_structured()` are routed there; all other arguments (including provider-specific arguments like `base_url`, `credentials`, or `api_args` for OpenAI-compatible endpoints) are passed to `ellmer::chat()`.
batch	logical; if `TRUE`, uses `ellmer::batch_chat_structured()` instead of `ellmer::parallel_chat_struct` Batch processing is more cost-effective for large jobs but may have longer turnaround times. Default is `FALSE`. See `ellmer::batch_chat_structured()` for details.
tools	Optional list of **ellmer** tool objects to register on the chat before coding: a provider's hosted web-search tool (`ellmer::openai_tool_web_search()`, `ellmer::claude_tool_web` `ellmer::google_tool_web_search()`) or a custom tool from `ellmer::tool()`. A single tool may be passed directly. Default is `NULL`, no tools. Tools change the instrument: with a hosted web search the model draws on live sources rather than its training data, so they are recorded on the object, disclosed by `print()` and `qlm_trail()`, carried to backfill passes and to a replication on the same endpoint (a provider's hosted tool belongs to that provider), and kept in the trail as name, type, description and configuration rather than as objects. Three limits. A hosted tool takes effect on both coding paths, but a custom tool only on the JSON path: the structured transport sends a custom tool's definition and runs no tool-calling loop, so the model may request it and get no result. Tools cannot be used with `batch = TRUE`, which does not send them. And a hosted tool's calls are billed by the provider outside token pricing, so the run's cost is tokens only, as its cost note says.
structured	character; how the output schema is obtained. `"structured"` sends it through the provider's structured-output mechanism. `"json"` asks for JSON, puts the schema in the system prompt, and re-prompts a unit whose response does not

conform. "auto" (the default) attempts the structured call and falls back to "json" if it fails, or if no response it completed matched the schema. On either path every response is validated against the codebook locally before it is tabulated. The JSON path asks for JSON syntax in the field the transport takes: text.format for native OpenAI's Responses API and response_format for every provider ellmer reaches through Chat Completions, including registered and openai_compatible/ endpoints. Providers with neither field, Anthropic among them, use prompted JSON with the same parsing, validation and repair. See Details for which to use.

json_retries — Integer; the number of additional requests **quallmer** may make for a unit on the JSON path after an unusable response. Default is 2, giving at most three JSON-path requests per unit. This is implemented by **quallmer** and is not passed to **ellmer**. Each request separately uses **ellmer**'s transport retry policy, controlled by options(ellmer_max_tries =). Applies on the JSON path only, so setting it alongside structured = "structured" is an error. What is still unusable after the run is left for backfill.

on_error — character; what a failed request does to the rest of a parallel call, passed to ellmer::parallel_chat_structured() or, on the JSON path, ellmer::parallel_chat(). "continue" (the default) attempts every unit and records each failure in the .error column, for qlm_failures() to list and qlm_backfill() to re-code. "return" stops submitting new requests after the first failure, waits for those in flight, and returns what the call has. On the structured path that call is the run: the units never sent are recorded in .error as not completed, for qlm_backfill() to send. On the JSON path the call is one wave: a unit the wave did not reach counts as unanswered, so json_retries sends it again in a later wave, each stopped in turn at its first failure, and whatever is still unsent at the end is recorded in .error; json_retries = 0 stops after the first wave. "stop" raises the first failure as an error. Applies to parallel runs only: the batch API has no equivalent, so it cannot be set with batch = TRUE.

backfill — Logical, integer or NULL; whether to complete the run before it is returned, by re-coding the units still failed with qlm_backfill(), using the same model and settings. FALSE or 0 (default) leaves the run as it came back; TRUE makes the default number of passes, currently two; a positive integer makes at most that many; NULL means FALSE here, since a fresh run has no parent whose passes could be replayed, which is what NULL asks qlm_replicate() for. Each pass is recorded in the object's metadata, and a pass that recovers nothing ends the backfill early. See qlm_backfill() for what is retried and what is left alone.

prices — Optional. Rates for costing the run when ellmer cannot: a named numeric vector or list with input and output, and optionally cached_input, in US dollars per million tokens, for example c(input = 0.435, output = 0.87, cached_input = 0.0036). Where ellmer prices the model itself its figure stands and these are not used. A cached_input rate that is not given is taken as the input rate. See the section on cost. Default is NULL.

name — character or NULL; a name identifying this coding run. Default is NULL.

notes — character or NULL; descriptive notes about this coding run. Useful for documenting the purpose or rationale when viewing results in qlm_trail(). Default is NULL.

Details

Arguments in ... are dynamically routed to either ellmer::chat(), ellmer::parallel_chat_structured(), or ellmer::batch_chat_structured() based on their names.

Progress indicators and error handling are provided by the underlying ellmer::parallel_chat_structured() or ellmer::batch_chat_structured() function. Set verbose = TRUE to see progress messages during coding. Retry logic for API failures should be configured through ellmer's options; what a failure does to the rest of a parallel run is on_error.

Value

A qlm_coded object (a tibble with additional attributes):

Data columns The coded results with a .id column for identifiers. A type_enum() declared "ordinal" in the codebook is an ordered factor whose levels are the enum's values in the order written; see the levels argument of qlm_codebook().

Attributes data, input_type, and run (list containing name, batch, call, codebook, chat_args, execution_args, metadata, parent).

The object prints as a tibble and can be used directly in data manipulation workflows. The batch flag in the run attribute indicates whether batch processing was used. The execution_args contains all non-chat execution arguments (for either parallel or batch processing).

Image input

An image codebook codes one image per element of x. A file path is read and sent inline, after being resized as the codebook's image_file_resize says: "high" by default, which fits the image within 2000x768 or 768x2000 pixels, "low" for 512x512, "none" to send the file as it is, or a magick geometry string. Anything but "none" needs the **magick** package, which is checked here before any request is sent. The resolution is part of the codebook because it is part of the measurement: a poster whose small print is legible at one size is not at another, and a replication should read the image the original run read. Codebooks saved before the setting existed are read as "low", which is what they were coded at. See qlm_codebook().

A URL is passed to the provider as it is, through ellmer::content_image_url(), so image_file_resize does not apply to it; what the provider does with a remote image is its own affair, and not every provider fetches URLs. The codebook's image_url_detail asks the provider for "low" or "high" detail on such an image, where the provider reads that field: OpenAI and OpenAI-compatible providers do, others ignore it, and ellmer forwards it only from the version that includes https://github.com/tidyverse/ellmer/pull/1133. When a value other than "auto" cannot take effect, qlm_code() says so before the run rather than recording a setting that was not applied. A path that does not exist is refused before anything is sent, so a URL typed without its scheme fails here with the path named, not inside the request.

Provider-specific parameters

params and api_args are forwarded to ellmer::chat() unchanged. quallmer does not inspect or rewrite either, so which of the two a setting belongs in is determined by ellmer and the provider, not here.

The distinction matters. `ellmer::params()` carries provider-agnostic settings that ellmer translates per provider; `api_args` goes into the raw request body untouched. A setting placed in the wrong one is not necessarily rejected. For OpenAI-compatible providers ellmer maps `top_k` onto the OpenAI field `top_logprobs`, which asks for log-probabilities per token and has nothing to do with top-k sampling — so `params(top_k = 20)` is rejected by Alibaba Model Studio (`Range of top_logprobs should be [0, 5]`), while `params(top_k = 3)` is accepted and silently applies no sampling setting at all. Non-OpenAI sampling controls therefore belong in `api_args`:

```
# Qwen through Alibaba Model Studio
qlm_code(
  x, codebook,
  model = "openai_compatible/qwen3-max",
  base_url = "https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
  credentials = function() {
    list(Authorization = paste("Bearer", Sys.getenv("DASHSCOPE_API_KEY")))
  },
  params = ellmer::params(temperature = 0.6, top_p = 0.95),
  api_args = list(top_k = 20, min_p = 0, enable_thinking = TRUE)
)

# Kimi K3 through Moonshot, whose temperature and top_p are fixed by the
# provider and documented as needing to be omitted rather than set
qlm_code(
  x, codebook,
  model = "openai_compatible/kimi-k3",
  base_url = "https://api.moonshot.ai/v1",
  credentials = function() {
    list(Authorization = paste("Bearer", Sys.getenv("MOONSHOT_API_KEY")))
  },
  api_args = list(reasoning_effort = "max")
)
```

Passing a model parameter such as `temperature` or `max_tokens` at the top level does not work: those reach `ellmer::chat()`, which has no such argument. Use `params`.

Cost

`include_tokens = TRUE` and `include_cost = TRUE` are forwarded to ellmer, which adds per-unit token counts and a `cost` column in US dollars. ellmer prices from a table fixed at its release, matched exactly on provider and model, and returns `NA` on any miss. Some providers are absent from that table altogether, DeepSeek among them, so no model of theirs is ever priced; a model newer than the installed ellmer is missed on a provider it otherwise prices, which upgrading fixes; and local endpoints such as ollama have no per-token charge. In each case `qlm_code()` says so once before the run, and the reason is kept with the object and shown when it is printed. With `include_tokens = TRUE` the token counts are recorded, from which such a run can be costed at the provider's published rates.

`prices` does that costing, at rates you supply from the provider's published price list. Supplying them implies `include_tokens = TRUE` and `include_cost = TRUE`. Only rows ellmer left `NA` are

filled, by the same sum ellmer applies to its own table: uncached input tokens at the `input` rate, cache hits at the `cached_input` rate, output at the `output` rate, each per million. Where ellmer priced every row itself the rates are not used, and you are told so. The rates are kept in the run's metadata, shown by `print()` and in the trail report, and reused by `qlm_replicate()` when the model, the endpoint, the batch setting and the service tier are unchanged, so a cost that rests on entered figures is always labelled as such. quallmer bundles no prices of its own.

Schema enforcement and validation

Some providers accept a JSON Schema without enforcing it, so a response can come back parseable but non-conforming: a number as a string, a required property missing, an extra one added. Providers reached through ellmer's generic OpenAI-compatible request path are all in this position: `strict = TRUE` is sent and may simply be ignored. Converted straight to a table, such a response would arrive silently as `NA`, or as an empty list-column cell that a valid empty answer also produces.

So every response is validated against `codebook$schema` before it is converted, on either path and whatever the provider: required properties present and not null, scalars of the declared type without coercion, enum values from the declared set, arrays and nested objects of the declared shape, no undeclared properties unless the schema allows them. A response that fails is a failed unit: its row is `NA`, its `.error` names the offending JSON path (`$.claims[2].score must be a number`), and `qlm_failures()` lists it for `qlm_backfill()` to re-code. The other units keep their coded values. The response's usage is recorded with the failure, since the request was billed.

`structured` chooses how the schema reaches the model, and what to do when the provider ignores it:

`"structured"` Send the schema through the provider's structured-output mechanism. Fails loudly if the call fails; a response that does not conform is a failed unit.

`"json"` Ask for JSON, put the schema in the system prompt, and re-prompt with the specific validation error when a response does not conform, up to `json_retries` times. The reliable choice for an endpoint known not to enforce.

`"auto"` Attempt the structured call; fall back to `"json"` for the whole run if it errors, or if every response the provider completed fails validation, which is what an endpoint that ignored the schema produces. Requests the provider refused, and responses it cut off or filtered, are left out of that judgement, since neither says anything about the schema. The fallback re-codes the units JSON mode can help: a response cut off at the output limit, or an input rejected as longer than the context window, fails the same way on any path, so those units keep the row, reason and usage the structured attempt gave them. A run in which some responses conform keeps them, and records the rest as failed units: the intermittent kind of non-enforcement is caught per unit, not by re-coding the corpus.

Where every completed response fails validation and nothing can fall back, under `"structured"`, `batch = TRUE` or a file input, the run is returned with every unit failed and its reason recorded, and a warning says why: the failed rows, their reasons and their usage are what the run has, and `qlm_backfill()` can retry them with another model.

On either path, `qlm_failures()` lists the units that produced no usable coding, with the reason for each, and `print()` reports how many there were. Most such failures are transient, and `qlm_backfill()` re-codes just those units and merges them back; `backfill` does that before returning. Batch processing and file inputs (image and audio codebooks) are not supported on the JSON path, so `"auto"` will not fall back under `batch = TRUE` or for a file input: a failed structured

call then stops with the provider's own error, and `structured = "json"` is refused up front. The path actually taken is recorded in the run metadata as `backend`, and a run validated this way carries `validation = "local"`.

The schema itself must be a `type_object()` at the root, whose properties become the columns of the result, built from `type_string()`, `type_boolean()`, `type_integer()`, `type_number()`, `type_enum()`, `type_array()` and `type_object()`. Any other type is refused before a request is sent, since there would be nothing to validate a response against.

Audio input

A codebook with `input_type = "audio"` codes recordings in one pass: each file in `x` is uploaded to the provider through ellmer's file upload, and the model receives a reference to it with the codebook's instructions, so the schema can ask for a transcript, a language, a summary or any coding of the content. Accepted formats are mp3, wav, ogg, m4a, flac and aac.

Which providers accept audio this way is not checked in advance: the recordings are uploaded and the model is asked, and a provider that cannot take them refuses with its own message, to which `qlm_code()` adds what is known. As of this version only Google Gemini (`google_gemini/`) is known to accept an uploaded recording alongside a schema-constrained request; OpenAI's and Anthropic's endpoints refuse, and Vertex has no file upload. For those providers, transcribe the recordings first with `qlm_transcribe()` and code the transcripts with a text codebook.

Every upload completes before the first coding request is sent, so either all the inputs are ready or nothing is spent; a failed upload stops the run with the provider's message, which says whether the failure was transient or the file itself. Uploads expire after 48 hours and storage is free, so `qlm_replicate()` and `qlm_backfill()` upload the files again from the paths in `x`. Before they do, they check the files against the SHA-256 hashes the run recorded for each unit, and refuse to continue if a path now points at different bytes. The hashes are taken before anything is uploaded, so they are of the bytes the model received even if a file is replaced while the requests run; they are kept in the run's metadata as `input_files` and reported by `qlm_trail()`. A backfill pass records its own hashes for the units it re-coded, so a run coded by an earlier version that recorded none gains them unit by unit; units still without one are reported as unverifiable, with a notice, rather than as changed.

`batch = TRUE` is not supported for audio: ellmer's batch cache is keyed on the prompts, and an upload gets a new reference every time, so a batch run could not be resumed.

With `include_cost = TRUE`, or `prices`, the cost of an audio run is potentially underestimated: providers charge more per audio token than per text token, and the figure is computed at the text rate from the total. The run's cost note says so, in `print()`, the trail, and any backfill pass.

Video input

A codebook with `input_type = "video"` codes picture and sound in one pass. Each element of `x` is one of: the path of a local video file (`mp4`, `mov`, `avi`, `wmv` or `webm`), which is uploaded to the provider; a YouTube link, which the provider fetches itself, so nothing is uploaded; or the URL of a video file, which is downloaded here and then uploaded like a file, and must end in one of those extensions. The model receives a reference to each with the codebook's instructions, so the schema can ask for a transcript, a description of what is shown, or any coding of the content. A video carries its audio track, so speech and picture are coded together.

As with audio, which providers accept video is not checked in advance; a provider that cannot take it refuses with its own message, and `qlm_code()` adds what is known. As of this version only Google Gemini (`google_gemini/`) accepts video, and all its chat models do. Gemini samples one frame a second and, at its default resolution, spends roughly 300 input tokens per second of video, so a ten-minute clip is about 175,000 tokens and a model with a million tokens of context takes about an hour of video per request. Before uploading, `qlm_code()` says how much it is about to send: the total duration and a token estimate when the **av** package is installed, the total size otherwise. A file over 2 GB, the upload's limit, is refused. Video tokens are charged at the text rate, so a cost from `include_cost` or `prices` needs no qualification here, unlike audio. A YouTube video must be public; the free tier caps YouTube input at eight hours of video a day. Gemini's own video settings (frame rate, clip offsets, media resolution) are not exposed by ellmer, so whole clips are coded at the defaults.

Provenance and replication work as for audio. The run records each file's size and SHA-256 hash; for a downloaded URL those are of the downloaded bytes, and for a YouTube link the URL alone is recorded, since nothing passed through this machine. `qlm_replicate()` and `qlm_backfill()` download and upload again, checking the hashes first, and `batch = TRUE` is refused as for audio.

Transcripts

A `qlm_transcript` from `qlm_transcribe()` is coded as ordinary text with a text codebook, on any provider. The run records the provenance of every transcript, the recording's hash, the transcription model, language, prompt and usage, in its metadata as `transcription`, and `qlm_trail()` reports it, so the trail documents the transcription as part of the instrument rather than starting at the text. `qlm_replicate()` and `qlm_backfill()` code the stored transcripts again without another transcription request, and carry the record forward.

A text unit that is `NA`, whether a transcription that failed or a missing value in any character vector, is never sent to the model. It is recorded as a failed unit with the reason, the transcription's own message where there is one, and `qlm_backfill()` leaves it alone, since there is nothing to retry. Transcribe the recording again and assign the text at that position before coding.

Rejected runs

When the provider rejects every request with a status that will not change on retry (400, 401, 403, 404 or 422), `qlm_code()` stops rather than returning a table of `NA`s. The most common cause is a model name the provider does not have, and providers rarely say so; many answer with a bare "HTTP 400 Bad Request". So before reporting, `qlm_code()` asks the provider for its model list, through ellmer's `models_<provider>()`, and says when the name is not on it, with the nearest names it does have. The lookup runs only after a run has failed, once per failed run, and only for providers whose listing is known to cover every name they will invoke (Bedrock, for one, invokes inference profiles its listing omits). Where it cannot run, or the name is on the list, the provider's own error is reported unchanged.

Under the default `on_error = "continue"` the parallel call does not stop at the first refusal, so every unit is sent once before the run comes back and is diagnosed. Each such request is refused before anything is generated, so it is cheap, but on a large corpus there are many of them, paced by `rpm`. `on_error = "return"` stops the structured call after the first wave, at the cost described under that argument; on the JSON path, whose default has always been `"continue"`, `json_retries` sends the units a wave did not reach in later waves, so `"return"` stops after the first wave there only with `json_retries = 0`.

Incomplete runs

A run over a corpus of any size rarely comes back complete, and an incomplete run is not an error. Under the default `on_error = "continue"` every unit is attempted, and a unit that produced no usable coding is returned as a row of `NA` with the reason in its `.error` column; `on_error` says what a failure does to the rest of the run. `qlm_failures()` lists the failed units with their reasons, and `print()` counts them. Trying again happens in layers. ellmer retries each request at the transport level, on every path. On the JSON path, `json_retries` sends a unit again during the run when its response was unusable. After the run, `backfill`, or `qlm_backfill()` on the returned object, re-codes what is still failed and merges it back, with the same model and settings. A backfill leaves alone the two failures that re-sending the same request cannot fix, a text rejected as longer than the context window and a response cut off at `max_tokens` (see below); those need a different `model` or a higher `params(max_tokens = )`. The section "When a run comes back incomplete" of the workflow guide (https://quallmer.github.io/quallmer/articles/pkgdown/getting-started/workflow.html) walks through this on a run that ships with the package, and its table matches each kind of failure to the mechanism that handles it.

Truncated responses

A response that runs into the provider's output limit (`max_tokens`) is cut off mid-JSON, and the request is billed in full. The affected units are systematically the longest and richest ones, and for a codebook where an empty answer is a legitimate outcome, a cut-off answer that reads as empty is the worst kind of silent failure.

The provider's finish reason travels with every response, on either path, and is read before the response is parsed: a unit cut off this way is recorded in `.error` with the token count, whether or not the fragment happens to parse, is listed by `qlm_failures()`, and is not retried: a repair prompt cannot supply what the limit withheld, and would only press the model into a shorter answer. A backfill leaves it alone too; raise `params(max_tokens = )` and backfill again. A response the provider withheld under a content filter, or finished for a reason it did not recognise, is recorded the same way with that reason.

When `batch = TRUE`, the function uses `ellmer::batch_chat_structured()` which submits jobs to the provider's batch API. This is typically more cost-effective but has longer turnaround times. The `path` argument specifies where batch results are cached, `wait` controls whether to wait for completion, and `ignore_hash` can force reprocessing of cached results. `on_error` does not apply: the batch API has no equivalent.

Registered providers

OpenAI-compatible endpoints can also be addressed by a registered prefix, for example `model = "dashscope/qwen-plus"`. See `qlm_register_provider()` for built-in endpoints, credential sources, and adding a private gateway. Native ellmer prefixes take precedence.

See Also

`qlm_codebook()` for creating codebooks, `qlm_replicate()` for replicating coding runs, `qlm_compare()` and `qlm_validate()` for assessing reliability.

Examples

```
# Requires API credentials and internet access; not run in package checks.
## Not run:
# Basic sentiment analysis
texts <- c("I love this product!", "Terrible experience.", "It's okay.")
coded <- qlm_code(texts, data_codebook_sentiment, model = "openai/gpt-4o-mini")
coded

# With named inputs (names become IDs in output)
texts_named <- c(review1 = "Great service!", review2 = "Very disappointing.")
coded2 <- qlm_code(texts_named, data_codebook_sentiment, model = "openai/gpt-4o-mini")
coded2

# Audio recordings, coded in one pass by a Gemini model; see the section
# "Audio input" for which providers accept audio. The model hears the
# recording, so the schema can ask for the transcript as well as the codes
speech_codebook <- qlm_codebook(
  "Speech", "Transcribe the recording, identify the language and summarise what is said.",
  ellmer::type_object(
    transcript = ellmer::type_string("Verbatim transcript, in the language spoken"),
    language = ellmer::type_string("Language spoken"),
    summary = ellmer::type_string("One-sentence summary in English")
  ),
  input_type = "audio"
)
coded_audio <- qlm_code(
  c(interview1 = "interview1.mp3", interview2 = "interview2.wav"),
  speech_codebook, model = "google_gemini/gemini-2.5-flash"
)

# A video codebook takes local files, YouTube links and URLs of video
# files in one vector; see "Video input" for what is uploaded and what
# the provider fetches itself
codebook_video <- qlm_codebook(
  name = "Video description",
  instructions = "Describe what is shown and transcribe what is said.",
  schema = ellmer::type_object(
    transcript = ellmer::type_string("Verbatim transcript of the speech"),
    setting = ellmer::type_string("Where the video is filmed, in a few words")
  ),
  input_type = "video"
)
coded_video <- qlm_code(
  c(clip = "clip.mp4", zoo = "https://www.youtube.com/watch?v=jNQXAC9IVRw"),
  codebook = codebook_video,
  model = "google_gemini/gemini-2.5-flash"
)

## End(Not run)
```

qlm_codebook *Define a qualitative codebook*

Description

Creates a codebook definition for use with qlm_code(). A codebook specifies what information to extract from input data, including the instructions that guide the LLM and the structured output schema.

Usage

```
qlm_codebook(
  name,
  instructions,
  schema,
  role = NULL,
  input_type = input_types(),
  levels = NULL,
  image_file_resize = NULL,
  image_url_detail = NULL
)
```

Arguments

name	Name of the codebook (character).
instructions	Instructions to guide the model in performing the coding task.
schema	Structured output definition, e.g., created by type_object(), type_array(), or type_enum().
role	Optional role description for the model (e.g., "You are an expert annotator"). If provided, this will be prepended to the instructions when creating the system prompt.
input_type	Type of input data: "text" (default), "image", "audio" or "video". For the other three the elements of x in qlm_code() are file paths; for "image" they may also be URLs, and for "video" YouTube links or URLs of video files. See the sections "Image input", "Audio input" and "Video input" of qlm_code() for how the files are handled and what is known about which providers accept them.
levels	Optional named list specifying measurement levels for each variable in the schema. Names should match schema property names. Values should be one of "nominal", "ordinal", "interval", or "ratio". If NULL (default), levels are auto-detected from schema types using the following mapping: type_boolean and type_enum = nominal, type_string = nominal, type_integer = ordinal, type_number = interval. A type_enum() is nominal unless declared "ordinal" here. For an ordinal enum the values, in the order they are written, are the scale order: type_enum(c("low", "medium", "high")) with levels = list(severity = "ordinal") ranks low

below medium below high. qlm_code() stores such a column as an ordered factor with those levels, and as_qlm_coded() does the same for human-coded data given this codebook, so qlm_compare() and qlm_validate() rank the categories by that order rather than alphabetically.

Names may refer to properties at any depth of the schema, including those inside a nested type_object() or the items of a type_array(). A level declared for a nested variable describes that variable in a table unnested to one row per item. qlm_compare() and qlm_validate() merge on .id, so such a table needs an identifier that is unique per item (a document-item key, not the document's .id alone) and must be re-wrapped with as_qlm_coded(), passing this codebook as codebook, for the declared levels to be found. Because levels is a flat list, a name that occurs at more than one place in the schema cannot be declared and is an error; rename the properties to make them distinct.

image_file_resize

How image files are resized before they are sent, for input_type = "image" only; setting it on any other codebook is an error. One of "high" (the default: fit within 2000x768 or 768x2000 pixels, whichever suits the orientation), "low" (fit within 512x512), "none" (send the file as it is), or a **magick** geometry string such as "1024x1024>". Anything other than "none" needs the **magick** package, and qlm_code() says so if it is missing. The value is stored on the codebook, so replications and backfills code at the same resolution and qlm_trail() records it. It applies to file paths only: an element of x that is a URL is fetched by the provider as it is. Codebooks saved before this field existed, including task() objects, are read as "low", the resolution they were coded at, rather than the current default. See ellmer::content_image_file().

image_url_detail

How much detail the provider should read from an image given as a URL, for input_type = "image" only: "auto" (the default, the provider chooses), "low" or "high". Passed to ellmer::content_image_url() for every element of x that is a URL; files are governed by image_file_resize instead. OpenAI and OpenAI-compatible providers use it and other providers ignore it, and ellmer forwards it only from the version that includes https://github.com/tidyverse/ellmer/pull/1133; qlm_code() says so before the run when a value other than "auto" cannot take effect. Stored on the codebook and recorded with the run like image_file_resize; codebooks saved before the field existed are read as "auto".

Details

This function replaces task(), which is now deprecated. The returned object has dual class inheritance (c("qlm_codebook", "task")) to maintain backward compatibility.

Value

A codebook object (a list with class c("qlm_codebook", "task")) containing the codebook definition. Use with qlm_code() to apply the codebook to data.

See Also

qlm_code() for applying codebooks to data, data_codebook_sentiment for a predefined codebook example, task() for the deprecated function.

Examples

```
# Define a custom codebook
my_codebook <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  )
)

# With a role
my_codebook_role <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  ),
  role = "You are an expert sentiment analyst."
)

# With explicit measurement levels
my_codebook_levels <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  ),
  levels = list(score = "interval", explanation = "nominal")
)

## Not run:
# Use with qlm_code() (requires API key)
texts <- c("I love this!", "This is terrible.")
coded <- qlm_code(texts, my_codebook, model = "openai/gpt-4o-mini")
coded

## End(Not run)
```

---

qlm_compare *Compare coded results for inter-rater reliability*

Description

Compares two or more coded objects to assess inter-rater reliability or agreement. For predefined-unit data (data frames or qlm_coded objects), computes standard reliability statistics. For segmented corpora from qlm_segment(), computes Krippendorff's alpha for unitizing (see Details).

Usage

```
qlm_compare(
  ...,
  by,
  level = NULL,
  tolerance = 0,
  ci = c("none", "analytic", "bootstrap"),
  bootstrap_n = 1000,
  by_category = FALSE
)
```

Arguments

...	Two or more data frames, qlm_coded, or as_qlm_coded objects to compare. These represent different "raters" (e.g., different LLM runs, different models, human coders, or human vs. LLM coding). Each object must have a .id column and the variable specified in by. Objects should have the same units (matching .id values). Plain data frames are automatically converted to as_qlm_coded objects. Alternatively, all inputs may be segmented corpora from qlm_segment() or as_qlm_coded() with qlm_segment = TRUE (see Details).
by	Optional. Name of the variable(s) to compare across raters (supports both quoted and unquoted). If NULL (default), all coded variables are compared. Can be a single variable (by = sentiment), a character vector (by = c("sentiment", "rating")), or NULL to process all variables.
level	Optional. Measurement level(s) for the variable(s). Can be: • NULL (default): Auto-detect from codebook • Character scalar: Use same level for all variables • Named list: Specify level for each variable Valid levels are "nominal", "ordinal", "interval", or "ratio".
tolerance	Numeric. Ratings agree when they differ by no more than tolerance. Default is 0, meaning numerical equality. The comparison allows a few units of floating-point rounding error scaled to the values being compared, so a difference of exactly tolerance counts as agreement and 0.1 + 0.2 equals 0.3; it is numerical equality rather than bit identity. The allowance is far smaller than any meaningful rating difference, so a genuinely different pair is never counted as agreeing. Used for percent agreement at ordinal, interval and ratio level. Ordinal categories given as text are ranked by their ordered-factor levels, and the tolerance then counts rank distance: tolerance = 1 means adjacent categories agree. Nominal categories agree only when identical.
ci	Confidence interval method:

"none" No confidence intervals (default)

"analytic" Analytic CIs where available (ICC, Pearson's r)

"bootstrap" Bootstrap CIs for all metrics via resampling

bootstrap_n Number of bootstrap resamples when ci = "bootstrap". Default is 1000. Ignored when ci is "none" or "analytic".

by_category Logical. If TRUE, include per-category reliability rows (alpha_per_value, kappa_per_value, and alpha_u_per_value for unitizing comparisons). Per-category rows are only produced for nominal-level data (or nominal-coded segments in unitizing comparisons); they are not meaningful for ordinal, interval, or ratio levels. Default is FALSE.

Details

The function merges the coded objects by their .id column and only includes units that are present in all objects. Missing values in any rater will exclude that unit from analysis.

Measurement levels and statistics:

- **Nominal**: For unordered categories. Computes Krippendorff's alpha, Cohen's/Fleiss' kappa, and percent agreement.
- **Ordinal**: For ordered categories. Computes Krippendorff's alpha (ordinal), weighted kappa (2 raters only), Kendall's W, Spearman's rho, and percent agreement.
- **Interval**: For continuous data with meaningful intervals. Computes Krippendorff's alpha (interval), ICC, Pearson's r, and percent agreement.
- **Ratio**: For continuous data with a true zero point. Computes the same measures as interval level, but Krippendorff's alpha uses the ratio-level formula which accounts for proportional differences.

Kendall's W, ICC, and percent agreement are computed using all raters simultaneously. For 3 or more raters, Spearman's rho and Pearson's r are computed as the mean of all pairwise correlations between raters.

How ratings are read. At ordinal, interval and ratio level every coder's ratings are read as numbers, whatever type they are stored as, so a column of digits held as text is compared on its values and not on the alphabetical order of its strings. A value that does not read as a number is an error at interval and ratio level, naming the coder and the values. At ordinal level the categories may instead all be text, in which case every coder's column must be an ordered factor with the same levels in the same order, and the categories are ranked by those levels; alphabetical order is never used, since it is not a ranking. A plain factor or a character column at ordinal level is an error that says how to declare the order: factor(x, levels = c(...), ordered = TRUE) for data built by hand, or a type_enum() written in scale order and declared "ordinal" in the codebook, which qlm_code() and as_qlm_coded() store as that ordered factor. Numbers from one coder and text from another cannot be placed on one scale and are an error. Nominal ratings are compared as text.

Per-category statistics. When by_category = TRUE and level = "nominal", the result also includes one row per category for alpha_per_value (from Krippendorff's alpha) and kappa_per_value (per-category kappa via dichotomisation for Cohen's, Fleiss' eq. 20–21 for Fleiss'). The marginal count n for each category is carried in the docid column. Per-category rows are not produced for ordinal, interval, or ratio levels.

Unitizing (segmentation) reliability [Experimental]

When all inputs are segmented corpora – created by `qlm_segment()` or `as_qlm_coded()` with `qlm_segment = TRUE` – agreement is measured at the character level using Krippendorff's alpha for unitizing continua (Krippendorff, 2019, section 12.6). This accounts for segments of unequal length and partial overlaps between coders' unitizations. The observed and expected coincidence matrices are constructed from the lengths of pairwise segment intersections across all observer pairs. The output includes a `docid` column with per-document and overall results. Segmented corpora must reference the same source text.

Four members of the unitizing alpha family are supported:

`alpha_u_binary` (`|_u`**alpha**) Computed when `by` is omitted. Measures agreement on which character spans are identified as segments versus gaps (irrelevant matter). Collapses all segment values to a binary distinction. Use this for pure boundary agreement when segments carry no codes (section 12.6.4, eq. 35).

`alpha_u_nominal` (`_u`**alpha[nominal**)] Computed when `by` names a docvar. Measures agreement on both boundary placement and the value (code) assigned to each segment. This is the most comprehensive measure: low values can reflect boundary disagreement, coding disagreement, or both (section 12.6.3, eq. 34).

`alpha_cu_nominal` (`_cu`**alpha[nominal**)] Computed alongside `alpha_u_nominal` when `by` is specified. Measures coding agreement *conditional on unitization*, restricting the coincidence matrix to intersections of non-gap segments only. This isolates "do the coders agree on the codes?" from "do they agree on the boundaries?" (section 12.6.5, eqs. 36–37).

`alpha_u_per_value[k]` (`_(k)u`**alpha[nominal**)] Computed alongside `alpha_u_nominal` when `by` is specified **and** `by_category = TRUE`. Reports the reliability of each individual value `k`, showing which codes are applied reliably and which are not. Coverage (the percentage of all `k`-valued matter found in valued intersections) is reported in the `docid` column (section 12.6.6, eq. 38).

Value

A `qlm_comparison` object (a tibble/data frame) with the following columns:

`variable` Name of the compared variable

`level` Measurement level used

`measure` Name of the reliability metric

`value` Computed value of the metric

`docid` Per-row context: source document identifier and overall indicator for unitizing comparisons; marginal (`n=X`) for nominal per-category alpha rows; `NA` otherwise.

`rater1`, `rater2`, ... Names of the compared objects (one column per rater)

`ci_lower` Lower bound of confidence interval (only if `ci != "none"`)

`ci_upper` Upper bound of confidence interval (only if `ci != "none"`)

The object has class `c("qlm_comparison", "tbl_df", "tbl", "data.frame")` and attributes containing metadata (`raters`, `n`, `call`).

Metrics by measurement level (predefined-unit comparisons):

- **Nominal:** alpha_nominal, kappa (Cohen's/Fleiss'), percent_agreement
- **Ordinal:** alpha_ordinal, kappa_weighted (2 raters only), w (Kendall's W), rho (Spearman's), percent_agreement
- **Interval/Ratio:** alpha_interval/alpha_ratio, icc, r (Pearson's), percent_agreement

For unitizing measures (segmented corpora), see Details.

Confidence intervals:

- `ci = "analytic"`: Provides analytic CIs for ICC and Pearson's r only
- `ci = "bootstrap"`: Provides bootstrap CIs for all metrics via resampling

References

Krippendorff, K. (2019). *Content Analysis: An Introduction to Its Methodology* (4th ed.). Sage. doi:10.4135/9781071878781

See Also

Related workflow functions: `qlm_validate()` for validation of coding against gold standards, `qlm_code()` for LLM coding, `as_qlm_coded()` for human coding, `qlm_segment()` for LLM-powered text segmentation.

Underlying reliability calculations (internal): `reliability_alpha()` and `reliability_alpha_u()` for Krippendorff's alpha; `reliability_kappa()` (Cohen) and `reliability_kappa_fleiss()`; `reliability_kendall_w()`; `reliability_icc()`.

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# Compare two coding runs
comparison <- qlm_compare(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  by = "sentiment",
  level = "nominal"
)
print(comparison)

# Compare specific variables with explicit levels
qlm_compare(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  by = "sentiment"
)
```

`qlm_failures` *List the units a coding run failed on*

Description

Reports which units of a `qlm_coded` object produced no usable coding, and why. A run over a real corpus rarely comes back complete: requests fail, providers refuse a text or reject it on length, and an endpoint can accept a schema and then ignore it. The object records all of this, in an `.error` list-column and as `NA` values, but nothing about its shape says how many units were affected, and for an array-valued property the obvious check does not work (see below). `print()` uses the same test to report a count.

Usage

```
qlm_failures(x)
```

Arguments

`x`	A `qlm_coded` object.

Details

A unit counts as failed when either of two things holds:

- it carries an `.error`. `qlm_code()` records one when the request failed, when the provider cut the response off or withheld it (see the *Truncated responses* section of `qlm_code()`), when the response held no JSON or JSON that did not parse, and when its JSON did not match the codebook schema, naming the offending path; or
- every required scalar property of the codebook schema is `NA` for it. That is how an object coded before every response was validated shows a response the endpoint sent without honouring the schema; a run coded since records such a unit under the first rule.

Array and nested-object properties are not consulted. After conversion, a missing array and a schema-valid empty one are the same zero-length list-column cell, so neither `is.na()` nor a row count on such a column can tell failure from a unit to which nothing applied. For a codebook whose required properties are all arrays or nested objects, only `.error` identifies failed units.

Value

A tibble with one row per failed unit and columns `.id`, `reason` (a character description) and `.error` (the recorded condition, or `NULL` for a unit that failed by returning `NA` for every required property). Zero rows when every unit was coded.

See Also

`qlm_backfill()` to re-code the failed units; `qlm_code()`, whose default `on_error = "continue"` attempts every unit and leaves the failed ones in the object rather than stopping the run; accessors for the other accessor functions.

Examples

```
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# A complete run: zero rows
qlm_failures(examples$example_coded_sentiment)

# A run that came back incomplete: a request that timed out, and responses
# cut off at max_tokens
qlm_failures(examples$example_coded_incomplete)

# The same run after qlm_backfill(): the timed-out unit recovered, the
# cut-off ones left alone, since re-sending the request cannot fix them
qlm_failures(examples$example_coded_backfilled)
```

`qlm_meta` *Get or set quallmer object metadata*

Description

Get or set metadata from `qlm_coded`, `qlm_codebook`, `qlm_comparison`, and `qlm_validation` objects. Metadata is organized into three types: user, object, and system. Only user metadata can be modified.

Usage

```
qlm_meta(x, field = NULL, type = c("user", "object", "system", "all"))

qlm_meta(x, field = NULL) <- value
```

Arguments

`x`	A quallmer object (`qlm_coded`, `qlm_codebook`, `qlm_comparison`, or `qlm_validation`).
`field`	Optional character string specifying a single metadata field to extract or set. If `NULL` (default), `qlm_meta()` returns all metadata of the specified type, and `qlm_meta<-()` expects `value` to be a named list.
`type`	Character string specifying the type of metadata to extract: `"user"` User-specified descriptive information (default). These fields are modifiable via `qlm_meta<-()`: `name` (run label) and `notes` (documentation). `"object"` Parameters defining how coding was executed. Read-only fields include: `batch`, `call`, `chat_args`, `execution_args`, `parent`, `n_units`, `input_type`. `"system"` Automatically captured environment information. Read-only fields include: `timestamp`, `ellmer_version`, `quallmer_version`, `R_version`. `"all"` Returns a named list combining all three types.
`value`	For `qlm_meta<-()`, the new value for the metadata field, or a named list of user metadata fields.

Details

Metadata is stratified into three types following the quanteda convention:

User metadata (`type = "user"`, default): User-specified descriptive information that can be modified via `qlm_meta<-()`. Fields: `name`, `notes`.

Object metadata (`type = "object"`): Parameters and intrinsic properties set at object creation time. Read-only. Fields vary by object type but typically include: `batch`, `call`, `chat_args`, `execution_args`, `parent`, `n_units`, `input_type`.

System metadata (`type = "system"`): Automatically captured environment and version information. Read-only. Fields: `timestamp`, `ellmer_version`, `quallmer_version`, `R_version`.

For `qlm_codebook` objects, user metadata includes `name` and `instructions` (the codebook instructions text), both of which can be modified.

Modification via `qlm_meta<-()` (assignment):

Only user metadata can be modified. For `qlm_coded`, `qlm_comparison`, and `qlm_validation` objects, modifiable fields are `name` and `notes`. For `qlm_codebook` objects, modifiable fields are `name` and `instructions`.

Object and system metadata are read-only and set at creation time. Attempting to modify these will produce an informative error.

Value

`qlm_meta()` returns the requested metadata (a named list or single value). `qlm_meta<-()` returns the modified object (invisibly).

See Also

- accessors for an overview of the accessor function system
- `codebook()` for extracting the codebook component
- `inputs()` for extracting input data

Examples

```
# Load example objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))
coded <- examples$example_coded_sentiment

# User metadata (default)
qlm_meta(coded)
qlm_meta(coded, "name")

# Object metadata
qlm_meta(coded, type = "object")
qlm_meta(coded, "call", type = "object")
qlm_meta(coded, "n_units", type = "object")

# System metadata
qlm_meta(coded, type = "system")
qlm_meta(coded, "timestamp", type = "system")
```

```
# All metadata
qlm_meta(coded, type = "all")

# Modify user metadata
qlm_meta(coded, "name") <- "updated_run"
qlm_meta(coded, "notes") <- "Analysis notes"

# Set multiple fields at once
qlm_meta(coded) <- list(name = "final_run", notes = "Final analysis")

## Not run:
# This will error - object and system metadata are read-only
qlm_meta(coded, "timestamp") <- Sys.time()

## End(Not run)
```

`qlm_register_provider` *Register an OpenAI-compatible endpoint*

Description

Register a provider prefix for use in `model = "provider/model"` with `qlm_code()` and `qlm_segment()`. Registration lasts for this R session. Native providers in the installed ellmer always take precedence.

Usage

```
qlm_register_provider(provider, base_url, api_key_env, overwrite = FALSE)
```

Arguments

`provider`	A lower-case prefix containing letters, digits, underscores or hyphens, starting with a letter.
`base_url`	An HTTP(S) API base URL, without credentials, a query or a fragment. Include the API path, for example `"https://example.org/v1"`.
`api_key_env`	Name of the environment variable holding the API key. ellmer reads its value when building the chat and making requests, and sends it as a bearer token.
`overwrite`	Replace an existing registration? Defaults to `FALSE`. Native ellmer providers cannot be replaced.

Details

Built-in prefixes are `dashscope` (Alibaba Model Studio, Singapore), `dashscope-cn` (Beijing), `moonshot` (Moonshot's international endpoint), and `zai` (Z.AI's general API, not its Coding Plan endpoint). Both DashScope prefixes read `DASHSCOPE_API_KEY`; Moonshot reads `MOONSHOT_API_KEY`, and Z.AI reads `ZHIPU_API_KEY`. Keys must belong to the endpoint's region. Model names are passed through

unchanged, including any slashes; the registry does not maintain a model catalogue, capability list or prices. Supply `prices` to `qlm_code()` when needed.

Use `batch = FALSE` with registered prefixes: ellmer currently has no batch submission method for its OpenAI-compatible provider. Registration supplies routing and credentials, not additional transport capabilities.

Explicit `credentials` (or ellmer's deprecated `api_key`) overrides the registered environment variable. Overriding `base_url` to a different URL requires explicit credentials, so a registered key cannot travel to another endpoint by accident. Use `credentials = function() ""` for an endpoint requiring no authentication.

Runs record the requested prefix and the effective `openai_compatible/model`, base URL and credential source. Replication and backfill reuse the recorded endpoint without consulting the registry. Supplying a replacement `model` resolves that model afresh. Registry entries contain environment variable names, never key values. A replication that overrides only the endpoint retains the original requested model, labelled "endpoint overridden" in print and trail output; its recorded `base_url` is the effective endpoint used for replay.

Value

The endpoint definition, invisibly.

See Also

`qlm_code()`, whose `model` argument takes a registered prefix.

Examples

```
qlm_register_provider("my_gateway", "https://example.org/v1", "GATEWAY_KEY")
## Not run:
# Another endpoint, using its own API key
qlm_register_provider(
  "minimax", "https://api.minimax.io/v1", "MINIMAX_API_KEY"
)
qlm_code(texts, codebook, model = "dashscope/qwen-plus")
qlm_code(texts, codebook, model = "my_gateway/organisation/model")

## End(Not run)
```

---

qlm_replicate *Replicate a coding task*

Description

Re-executes a coding task from a `qlm_coded` object, optionally with modified settings. If no overrides are provided, uses identical settings to the original coding: both the execution arguments and the arguments the original run passed to `ellmer::chat()`, such as `params` and `api_args`. Credentials and endpoint settings are an exception, and are carried over only while the endpoint itself is unchanged.

Usage

```
qlm_replicate(
  x,
  ...,
  codebook = NULL,
  model = NULL,
  batch = NULL,
  backfill = NULL,
  name = NULL,
  notes = NULL
)
```

Arguments

x	A qlm_coded object.
...	Optional overrides passed to qlm_code(), such as params, api_args, or max_active. Any setting not overridden is restored from the original run when the endpoint is unchanged, including the arguments it passed to ellmer::chat(). An endpoint is identified by both the provider prefix and base_url (an explicit base_url = NULL, meaning the provider's default host, counts as a change of endpoint), since every provider ellmer has no chat_*() for is reached as openai_compatible/<model> — so Qwen through Alibaba Model Studio and Kimi through Moonshot share a prefix while being different services with different credentials. When either changes, only portable chat settings (params and echo) are carried over; supply credentials, endpoint settings and other endpoint-specific arguments explicitly. An informational message names inherited arguments that were omitted and not explicitly replaced. Registered tools are carried like the rest: kept on the same endpoint, since a hosted tool belongs to its provider, and dropped with the message when it changes; pass tools to replace them. An object read back from a trail records its tools by description and configuration only, and those are not sent either.
codebook	Optional replacement codebook. If NULL (default), uses the codebook from x.
model	Optional replacement model (e.g., "openai/gpt-4o"). If NULL (default), uses the model from x.
batch	Optional logical to override batch processing setting. If NULL (default), uses the batch setting from x. Set to TRUE to use batch processing or FALSE to use parallel processing, regardless of the original setting.
backfill	Logical, integer, or NULL; controls backfilling after the replication. NULL (default) replays the passes recorded on x, using the same models and overrides in the same order. FALSE or 0 performs no backfill. TRUE runs a fresh backfill with the replication's model and the default number of passes, currently two; a positive integer runs at most that many fresh passes. A fresh backfill does not reproduce a different model used by the parent's recorded passes. If a replayed pass fails outright, the replication and any earlier recoveries are retained, the failure is recorded, and no later passes are replayed.
name	Optional name for this run. If NULL, defaults to the model name (if changed) or "replication_N" where N is the replication count.

notes — Optional character string with descriptive notes about this replication. Useful for documenting why this replication was run or what differs from the original. Default is NULL.

Details

The coding path is reproduced from the path the original run actually took, not the structured mode it requested: a run that asked for "auto" and fell back to JSON mode replicates as "json", so that an intermittently conforming endpoint cannot quietly skip the local validation the original relied on. Pass structured explicitly to override. When the endpoint changes, provider or base_url, the path is chosen afresh for it. By the same rule, a parent that was completed with qlm_backfill() has its passes replayed on the replication, so the two are complete on the same terms; see backfill.

Value

A qlm_coded object with run$parent set to the parent's run name.

See Also

qlm_code() for initial coding, qlm_compare() for comparing replicated results, qlm_backfill() to re-code only the units a run failed on.

Examples

```
## Not run:
# First create a coded object
texts <- c("I love this!", "Terrible.", "It's okay.")
coded <- qlm_code(texts, data_codebook_sentiment, model = "openai/gpt-4o-mini", name = "run1")

# Replicate with same model
coded2 <- qlm_replicate(coded, name = "run2")

# Compare results
qlm_compare(coded, coded2, by = "sentiment", level = "nominal")

## End(Not run)
```

---

qlm_segment *Segment texts using an LLM*

Description

Applies a codebook to input texts to segment them into thematic or conceptual units, returning a quanteda::corpus() where each segment is a document. This is the LLM-powered analogue of quanteda::corpus_segment().

Usage

```
qlm_segment(x, codebook, model, ..., prices = NULL, name = NULL, notes = NULL)
```

Arguments

x	A character vector of texts or a quanteda::corpus() object. Named character vectors use names as document identifiers; unnamed vectors use sequential labels (text1, text2, ...).
codebook	A codebook object created with qlm_codebook(). The schema should be a type_object() whose fields become docvars in the output corpus. Do not include a field named text; it is reserved for the verbatim segment text and is added automatically.
model	character; the provider (and optionally model) name in the form "provider/model" or "provider" (which will use the default model for that provider). Native prefixes are passed to ellmer::chat(). Registered prefixes, such as "dashscope/qwen-plus", resolve through qlm_register_provider() and require an explicit model name. Examples: "openai/gpt-4o-mini", "anthropic/claude-3-5-sonnet-20241022", "ollama/llama3.2", "openai" (uses default OpenAI model).
...	Additional arguments passed to ellmer::chat() or ellmer::parallel_chat_structured(). Arguments recognized by ellmer::parallel_chat_structured() are routed there; all other arguments (including provider-specific arguments like base_url, credentials, or api_args for OpenAI-compatible endpoints) are passed to ellmer::chat().
prices	Optional. Rates for costing the run when ellmer cannot: a named numeric vector or list with input and output, and optionally cached_input, in US dollars per million tokens. As for qlm_code(); see the section on cost there. Supplying them implies include_tokens = TRUE and include_cost = TRUE. Default is NULL.
name	character or NULL; a name identifying this coding run. Default is NULL.
notes	Optional character string with descriptive notes about this segmentation run. Default is NULL.

Details

The codebook schema defines additional document-level variables (docvars) for each segment. A text field (the verbatim segment text) is always added automatically and must not appear in the schema. Measurement levels defined in the codebook are not applicable to segmentation and are silently ignored.

Value

A quanteda::corpus() where each segment is a document. Document names follow the {source}.{i} convention of quanteda::corpus_segment(). Docvars include:

docid Name of the source document.

segid Integer segment index within the source document.

... Any fields defined in the codebook schema.

input_tokens, output_tokens, cached_input_tokens, cost With include_tokens = TRUE or include_cost = TRUE: the usage of the call made for the source document, repeated on each of its segments. See the section on cost.

... Original docvars inherited from the input (if x is a corpus).

The corpus metadata (see quanteda::meta()) carries name, continuum_lengths, and, when usage was requested, usage, a data frame with one row per source document, plus cost_note and prices where they apply.

Cost

Each source document is one request, so token counts and cost belong to the document, not to a segment. With include_tokens = TRUE or include_cost = TRUE they are recorded twice: in the corpus metadata as usage, one row per input document including documents that yielded no segments, and on each segment as docvars for convenience. Sum the metadata table for the run's total; summing the docvars counts a document once per segment, and a document that produced no segments has no docvars at all.

ellmer prices from a table fixed at its release and returns NA for a model it does not list; qlm_segment() says so once before the run, as qlm_code() does, and prices costs the run from the token counts at rates you supply. The four usage names are reserved when usage is requested: a codebook field or an inherited docvar of the same name is an error rather than silently overwritten.

See Also

qlm_code() for document-level coding, qlm_codebook() for creating codebooks, quanteda::corpus_segment() for pattern-based segmentation.

Examples

```
## Not run:
# Aspect-based segmentation of a hotel review (character vector input
# returns a data.frame).
review <- paste(
  "The room was clean and tidy, despite being rather basic in its furnishings.",
  "The location of the hotel was really great, however.",
  "We loved the proximity to both public transport and to the city's main attractions."
)

cb_absa <- qlm_codebook(
  name = "Aspect-based segmentation",
  instructions = paste(
    "Segment the text according to the distinct aspects (topics or features).",
    "Each segment will continue as long as it is part of the same aspect.",
    "An aspect-based segment may be more than one sentence or may be just a",
    "part of a sentence.",
    "",
    "Aspects in hotel reviews include: cleanliness, features, location, service,",
    "and value. Return each aspect segment with its verbatim text and a short",
    "aspect label."
  ),
  schema = type_object(
    aspect    = type_string("Short aspect label"),
    sentiment = type_enum(c("negative", "neutral", "positive"),
                          "Sentiment toward this aspect")
```

```
  )
)

segs <- qlm_segment(review, cb_absa, model = "anthropic")
quanteda::docvars(segs)
#   docid segid      aspect sentiment
# 1 text1     1 cleanliness  positive
# 2 text1     2    features  negative
# 3 text1     3    location  positive

# Corpus input preserves existing docvars
reviews_corp <- quanteda::corpus(
  c(hotel_a = review),
  docvars = data.frame(city = "London", stars = 4L)
)
segs_corp <- qlm_segment(reviews_corp, cb_absa, model = "anthropic")
quanteda::docvars(segs_corp)

## End(Not run)
```

`qlm_trail` *Create an audit trail from quallmer objects*

Description

Creates a complete audit trail documenting your qualitative coding workflow. Following Lincoln and Guba's (1985) concept of the audit trail for establishing trustworthiness in qualitative research, this function captures the full decision history of your AI-assisted coding process.

Usage

```
qlm_trail(..., path = NULL)
```

Arguments

`...`	One or more quallmer objects (`qlm_coded`, `qlm_comparison`, or `qlm_validation`). When multiple objects are provided, they will be used to reconstruct the complete workflow chain.
`path`	Optional base path for saving the audit trail. When provided, creates `{path}.rds` (complete archive) and `{path}.qmd` (human-readable report). If `NULL` (default), the trail is only returned without saving.

Details

Lincoln and Guba (1985, pp. 319-320) describe six categories of audit trail materials for establishing trustworthiness in qualitative research. The quallmer package operationalizes these for LLM-assisted text analysis:

Raw data Original texts stored in coded objects

Data reduction products Coded results from each run

Data reconstruction products Comparisons and validations

Process notes Model parameters, timestamps, decision history

Materials relating to intentions Function calls documenting intent

Instrument development information Codebook with instructions and schema

When `path` is provided, the function creates:

- `{path}.rds`: Complete trail object for R (reloadable with `readRDS()`)
- `{path}.qmd`: Quarto document with full audit trail documentation

Credentials: Both files are written to be shared, so neither carries the value of a credential a run was configured with. An `api_key`, the values of `api_headers` entries named like a credential, and any userinfo or credential-named query parameter in a `base_url` are replaced by `"<redacted>"` in each run's recorded call and chat arguments. The returned object is redacted in the same way, so the trail in memory and the two files agree. The trail records that a credential was supplied, not what it was; a `qlm_coded` object loaded from the `.rds` therefore needs a credential of its own before it can be replicated.

In a recorded call, a credential argument is kept only when it names a source that cannot itself contain the value: a variable, a qualified name, or an exact one-argument `Sys.getenv("MY_KEY")` lookup. Other computed expressions are replaced wholesale because their unevaluated arguments may contain a literal credential. The exact `credentials = function() Sys.getenv("MY_KEY")` callback is also kept, rebuilt without its environment; a callback of any other shape is replaced by `"<redacted>"`, since it may hold or capture the secret it returns.

Value

A `qlm_trail` object containing:

runs List of run information with coded data, ordered from oldest to newest

complete Logical indicating whether all parent references were resolved

References

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic Inquiry*. Sage.

See Also

`qlm_code()`, `qlm_replicate()`, `qlm_compare()`, `qlm_validate()`

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# View audit trail from two coding runs
trail <- qlm_trail(
  examples$example_coded_sentiment,
```

```
  examples$example_coded_mini
)
print(trail)


# Save complete audit trail (creates .rds and .qmd files)
qlm_trail(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  path = tempfile("my_analysis")
)
```

`qlm_transcribe` *Transcribe audio recordings*

Description

Transcribes audio files with a speech-to-text model and returns the transcripts as a character vector that carries the provenance of each one: the file it came from and its hash, the model, the language and prompt given, the time, and the usage the provider reported. Passed to `qlm_code()` with a text codebook, the transcripts are coded as ordinary text and that provenance is recorded with the run, so `qlm_trail()` documents the transcription as part of the measurement instrument and the same text can be coded again by `qlm_replicate()` or `qlm_backfill()` without another transcription request.

Usage

```
qlm_transcribe(
  x,
  model = "openai/gpt-4o-mini-transcribe",
  language = NULL,
  prompt = NULL,
  api_key = NULL,
  base_url = NULL,
  max_active = 10,
  rpm = 60,
  on_error = c("continue", "return", "stop"),
  ...
)
```

Arguments

`x`	character; paths of audio files, or `http(s)` URLs of them, optionally named. See the section "Names and identifiers".
`model`	character; the transcription model in `"provider/model"` form. See the section "Routes".

language	character; the ISO 639-1 code of the language spoken, such as "en" or "zh", or NULL to let the model detect it. Sent to the OpenAI endpoint as its language field; added to the instruction for a Gemini model.
prompt	character; optional text to guide the transcription, such as names and terms the recording contains or the style of punctuation wanted. Sent to the OpenAI endpoint as its prompt field; added to the instruction for a Gemini model.
api_key	character; the API key. NULL reads the environment variable the provider uses, OPENAI_API_KEY, the variable a registered provider was given, or the one ellmer reads for a chat provider. On the chat route the value is passed to ellmer as the credential itself, which is what Gemini and Anthropic take. The key is never recorded.
base_url	character; the endpoint to send the requests to. On the endpoint route, the prefix before /audio/transcriptions; NULL is the provider's own host. On the chat route, passed to ellmer's chat constructor. Recorded with any credential it carries redacted.
max_active	integer; the number of requests in flight at once, as in ellmer::parallel_chat().
rpm	integer; the request rate in requests per minute. The default is below ellmer's because transcription endpoints have their own, lower, rate limits. A rate-limited request is retried after the delay the provider asks for.
on_error	character; what to do when a transcription fails. See the section "Failures".
...	Reserved; must be empty.

Details

This is the two-stage route to audio: transcribe once, then code the text with any provider. The single-pass route, a codebook with input_type = "audio", sends the recording itself to a model that can hear it; see the "Audio input" section of qlm_code() for the providers that accept it.

Value

A named character vector of class qlm_transcript, one element per element of x in the same order, with attribute provenance, a data frame with one row per element:

.id the element's name.

status "ok", "failed" or "unsubmitted".

source the basename of a local file, or the URL with any credential it carried redacted.

.error the failure message, or NA.

size, sha256 the bytes transcribed and their hash; NA when a download failed.

model as given.

language, prompt as given, or NA.

base_url the host the requests went to, redacted: on the endpoint route always, on the chat route when given.

timestamp when the response arrived, or NA.

usage a list column holding what the provider reported, or on the chat route ellmer's tokens, cost and version.

Subsetting with `[`, renaming with `names<-` and concatenating with `c()` keep the table aligned with the elements. Assigning a `qlm_transcript` with `[<-` or `[[<-` replaces rows of the table too, so a retried transcription replaces the failure it retries; assigning plain text records an edit. `as.character()` drops the table.

Routes

The route is chosen from the provider prefix of `model`, not from a list of models known to transcribe. Whether a model can is for the provider to say, and asking costs nothing: an upload is free and a refused request is not billed.

- **The transcription endpoint**, `/audio/transcriptions`, for `openai/` and for any provider registered with `qlm_register_provider()`, at the host it was registered with. OpenAI's models are `gpt-4o-mini-transcribe` (the default), `gpt-4o-transcribe` and `whisper-1`; a registered host serves whatever it serves, such as Whisper on Groq. Each file must be at most 25 MB and one of `flac`, `mp3`, `mp4`, `mpeg`, `mpga`, `m4a`, `ogg`, `wav` or `webm`. OpenAI reports usage as audio and text tokens for the `gpt-4o` models and as seconds of audio for `whisper-1`.
- **A chat model that hears the recording**, for every other provider ellmer reaches: the file is uploaded through ellmer's file upload and the model is asked for a verbatim transcript. Known to work: Google Gemini's `pro`, `flash` and `flash-lite` models (`google_gemini/`). Anthropic takes no audio, and OpenAI's chat models refuse it; a provider that cannot take the recording says so in the failure recorded for each unit. Usage is the token count and the cost ellmer computes, with the qualification that ellmer prices audio tokens at the text rate. Gemini's dedicated transcription model, `gemini-3.5-transcribe`, cannot be reached through ellmer yet.

No dollar cost is computed on the endpoint route: ellmer has no rates for transcription models, and per-minute pricing does not fit a per-token table. The usage is recorded as reported so it can be costed by hand.

Names and identifiers

The names of the result become the `.id` of each unit when it is coded, and the document names when the vector is made a corpus. A supplied name is kept exactly. An unnamed local file is named by its basename; an unnamed URL is named `text1`, `text2`, ... by its position in `x`. The resolved names must be unique and non-empty: two files that share a basename need names supplied, and the error says so.

Failures

Requests run in parallel. Under `on_error = "continue"` every file is attempted and the result has an element for each, `NA` where the transcription failed, with the provider's message in the `.error` column of the provenance table. `"return"` stops submitting after the first failure and marks the files it never sent as such; `"stop"` raises the first error. A failed download, a failed upload on the chat route, a refused request and a response with no transcript in it are all failures of the unit, under the same policy. The one limit is on the chat route, where an empty answer is known only after every request has returned, so `"return"` cannot withhold submissions on its account. Validation of the arguments, the files and the model all happen before anything is downloaded or sent, and abort whatever `on_error` says.

A missing transcript passed to `qlm_code()` is never sent to the model: its unit is recorded as failed with the transcription's reason, and `qlm_backfill()` leaves it alone. Transcribe the file again and assign the result at that position, `transcripts[failed] <- qlm_transcribe(files[failed])`, which replaces the record with it; or concatenate independent runs with `c()`.

URLs

An element of `x` that is an `http://` or `https://` URL is downloaded to a temporary file, which is removed when the function returns. The hash and size recorded are those of the downloaded bytes, and the URL is recorded, with any credential it carried redacted, as the `source`. The format is read from the URL's path, so a URL with no file extension is refused before anything is fetched.

See Also

`qlm_code()` for coding the transcripts, and its "Audio input" section for the single-pass route; `qlm_trail()` for the record a coded transcript leaves.

Examples

```
## Not run:
files <- list.files("recordings", pattern = "\\.wav$", full.names = TRUE)
transcripts <- qlm_transcribe(files)
transcripts
attr(transcripts, "provenance")

# Code the transcripts with any provider; the run records the transcription
coded <- qlm_code(transcripts, codebook_sentiment, model = "anthropic/claude-sonnet-5")
qlm_trail(coded, path = "sentiment_trail")

# A Gemini chat model as the transcriber, with a language hint
transcripts <- qlm_transcribe(files, model = "google_gemini/gemini-2.5-flash",
                              language = "fr")

# Whisper on a registered OpenAI-compatible host
qlm_register_provider("groq", "https://api.groq.com/openai/v1", "GROQ_API_KEY")
transcripts <- qlm_transcribe(files, model = "groq/whisper-large-v3")

# A recording on the web, named so the name becomes its .id
url <- c(harvard = "https://www.voiptroubleshooter.com/open_speech/american/OSR_us_000_0010_8k.wav")
qlm_transcribe(url)

## End(Not run)
```

---

qlm_translate *Translate texts*

Description

[Experimental]

Usage

```
qlm_translate(
  x,
  model,
  to = "en",
  from = NULL,
  instructions = NULL,
  context = NULL,
  glossary = NULL,
  protect = NULL,
  params = NULL,
  chunk_size = NULL,
  adapter = NULL,
  api_key = NULL,
  base_url = NULL,
  max_active = NULL,
  rpm = 60,
  on_error = c("continue", "return", "stop"),
  ...
)
```

Arguments

x	Character vector of texts, optionally named, or a quanteda corpus. A transcript from qlm_transcribe() or an earlier translation can also be translated, and its history is kept. Names must be unique and non-empty. Missing texts are recorded as failures without a request.
model	The provider and model, as "provider/model"; required. See "Providers" for the translation services; any other ellmer::chat() model translates through qlm_code().
to	Target language, as a BCP 47 language tag such as "en", "en-GB", "pt-BR" or "zh-Hant". See "Language codes".
from	Source language, as a BCP 47 language tag, or NULL for the provider to detect it. Set it whenever you know it: see "Unsupported languages".
instructions	Character vector of translation directives, such as a register to use or how to treat names.
context	One string of background about the texts, to inform the translation but not to be translated.
glossary	Named character vector of source terms and their required translations, such as c(Bundestag = "Bundestag").
protect	Character vector of regular expressions matching text to keep exactly as it is, such as names or codes. Each is matched against the whole text, so a match may span paragraphs and ^ means the start of the text. Matches must not overlap or be empty; every one is checked in the translation.
params	Generation settings from ellmer::params(), such as temperature or max_tokens.

chunk_size	The most source characters in one request, or NULL for the provider's default. Characters only estimate tokens: a response cut off at the output limit is detected and sent again in smaller parts.
adapter	NULL to choose how requests are made from model, or "generic" to send a recognised translation model the ordinary chat request instead.
api_key	An API key, or NULL to use the provider's environment variable or saved key (see "API keys"). Keys are never recorded.
base_url	Optional API base URL, recorded with any credentials in it removed.
max_active	Maximum concurrent requests, or NULL for the provider's default.
rpm	Maximum requests per minute.
on_error	Failure policy: "continue" attempts every text, "return" stops sending after a failure, and "stop" raises it.
...	Named arguments specific to the provider; see "Providers". Any other name is an error.

Details

Translates texts and returns them as a named character vector carrying per-document translation provenance, as qlm_transcribe() does for transcripts. The result can be passed to qlm_code(); qlm_trail() then records translation as part of the coding instrument.

Providers:

to, from and protect work the same for every provider. The other arguments depend on it; where a provider cannot honour one, using it is an error rather than being ignored.

	DeepL	Qwen-M
model	"deepl/" and quality_optimized, latency_optimized or prefer_quality_optimized	"dashsc
instructions	at most 10, of at most 300 characters each	yes
context	yes	yes
glossary	error: not yet implemented	yes
params	error	tempera
...	formality: "default", "more", "less", "prefer_more" or "prefer_less"	none
One request	up to 50 whole documents	adjacent
chunk_size default	16,000	4,000
max_active default	2	2
API key	DEEPL_API_KEY, or keyring "deepl"	DASHSC
Also recorded	detected language, model type used, billed characters	model, t

DeepL model types express a quality or speed goal, not a fixed model version. Google's region selects the model's location, but the Basic API uses a global endpoint, so it does not guarantee where data is processed. For chat models the answering model version is not yet recorded (#199). Arguments passed to a chat model through ... are recorded by name only, since they can carry credentials.

Language codes:

`to` and `from` take BCP 47 language tags: an ISO 639 language code (`"fr"`), optionally followed by an ISO 15924 script (`"zh-Hant"`) and an ISO 3166 region or UN M.49 area (`"pt-BR"`, `"es-419"`). A plain language code is usually what you want; add a region only for a regional variant of the target. W3C's Choosing a language tag explains the parts, and the Language Subtag Lookup finds the tag for a language. Case does not matter, and an underscore may be used for a hyphen.

Each provider spells some languages differently, so the tag is converted to the provider's own code, and the provenance records both (`settings$provider_languages`). For Chinese, an explicit script decides between Simplified and Traditional (`zh-Hans-HK` is Simplified); a region only supplies the usual script when none is given (`zh-TW` is Traditional).

- DeepL takes regional variants only as targets, and only `en-GB`, `en-US`, `fr-CA`, `fr-FR`, `de-DE`, `de-CH`, `pt-BR`, `pt-PT`, `es-ES`, `es-419`, `zh-Hans` and `zh-Hant`; any other is an error before any request. A source is reduced to its language. `zh-CN` and `zh-TW` become `ZH-HANS` and `ZH-HANT`, `no` becomes `NB`, and `fil` becomes `TL`.
- Google Translation LLM: `zh-CN` and `zh-TW` become `zh-Hans` and `zh-Hant`, and `tl` becomes `fil`.
- Qwen-MT takes language codes only, with `zh_tw` for Traditional Chinese; a regional variant such as `en-GB` is requested in its instructions instead.
- Chat models receive the tag as given.

Unsupported languages:

Language detection fails silently. Given text in a language it does not support, every provider tested, DeepL, Google, Qwen-MT and a chat model, returned a fluent but invented English text, recorded as `"ok"`; DeepL and Google reported a wrong detected language (Swahili for Kimbundu). With `from` set, DeepL, Google and Qwen-MT instead refuse an unsupported language, and the text is `NA` with the provider's reason. Chat models do not refuse, and may invent a translation even when told the source language.

So set `from` whenever the source language is known, and when it is not, check the recorded `detected_language` against what you expect. For a corpus in several languages, translate each language with its own call and combine the results with `c()`. Results that look fluent are not evidence of a correct translation.

API keys:

- DeepL: see access and authentication. A key ending in `:fx` uses the Free API host.
- Qwen-MT: see Model Studio API keys. The key must match the endpoint's region.
- Google: see Cloud Translation setup. Enable billing and the Cloud Translation API for the project, and create an API key allowed to call it.
- Chat models: see the ellmer provider documentation.

Where the table names a keyring entry, a key saved once with, for example, `keyring::key_set("deepl")` is used when the environment variable is unset; this needs the optional **keyring** package and may prompt to unlock the system store. An explicit `api_key` overrides both.

Paragraphs, retries and failures:

Paragraphs are separated by blank lines. The separators, and whitespace at the start and end of each paragraph, are kept aside and restored exactly. DeepL, Qwen-MT and chat models are told where each paragraph begins, and a response that loses, merges or reorders paragraphs is sent again in smaller parts. Google receives each document whole and handles paragraph formatting itself, so exact separators are not guaranteed there. Text longer than `chunk_size` is split at sentence ends, else at spaces, else between characters.

A response that is malformed or changes a protected term is retried once, and one cut off at the output limit is sent again in smaller parts. A document with any part that still fails is NA, never a partial translation, and its error says why. Every attempt, with its usage, is recorded in the provenance (attempts). The provenance records the settings requested, not a guarantee that the provider applied them. Save the result with saveRDS() to keep the exact translations used for coding.

Value

A qlm_translation: a character vector of translations named by document, with NA for each translation that failed or was not submitted. Its "provenance" attribute is a data frame with one row per text: the source text and its hash, the output hash, requested and reported models, languages, adapter, settings, status, error, timestamp, any earlier preprocessing history, and attempts: for each text, a data frame with one row per request attempt, giving the segment, any error, the reductions made, and usage (tokens and cost, or DeepL's billed characters).

Subsetting with [, renaming, and c() keep the provenance aligned. Assigning plain text records an edit; assigning another qlm_translation, such as a retry of failed texts, carries its provenance. as.character() drops the provenance, and inputs() returns the source texts.

For a corpus input, the document variables and metadata are kept, and quanteda::corpus(x) (see corpus.qlm_translation()) returns a corpus of the translated texts with them restored.

Examples

```
## Not run:
# Reads DEEPL_API_KEY, or supply api_key = keyring::key_get("deepl")
translated <- qlm_translate(c(speech = "Bonjour tout le monde."),
  model = "deepl/quality_optimized", to = "en-GB")
coded <- qlm_code(translated, data_codebook_sentiment,
  model = "openai/gpt-4o-mini")
qlm_trail(coded, path = "translation_trail")

texts <- c(fr = "Bonjour tout le monde.", de = "Guten Morgen.")

# Qwen-MT: reads DASHSCOPE_API_KEY for the Singapore endpoint
translated_qwen <- qlm_translate(texts,
  model = "dashscope/qwen-mt-plus", to = "en-GB",
  context = "Greetings in everyday conversation.",
  params = ellmer::params(temperature = 0.1))
translated_qwen

# Google: reads GOOGLE_TRANSLATE_API_KEY, falling back to the optional
# keyring entry saved with keyring::key_set("google-translate")
translated_google <- qlm_translate(texts,
  model = "google-translate/translation-llm", to = "en-GB",
  project_id = "your-google-cloud-project-id")
# Alternatively, set GOOGLE_CLOUD_PROJECT and omit project_id.
attr(translated_google, "provenance")

# DeepL with a formal register
qlm_translate(texts, model = "deepl/quality_optimized", formality = "more")
```

```
# Convert to a quanteda corpus, if quanteda is installed
quanteda::corpus(translated_google)

## End(Not run)
```

`qlm_validate` *Validate coded results against a gold standard*

Description

Validates LLM-coded results from one or more `qlm_coded` objects against a gold standard (typically human annotations) using appropriate metrics based on measurement level. For nominal data, computes accuracy, precision, recall, F1-score, and Cohen's kappa. For ordinal data, computes accuracy and weighted kappa (linear weighting), which accounts for the ordering and distance between categories.

Usage

```
qlm_validate(
  ...,
  gold,
  by,
  level = NULL,
  average = c("macro", "micro", "weighted", "none"),
  ci = c("none", "analytic", "bootstrap"),
  bootstrap_n = 1000
)
```

Arguments

`...`	One or more data frames, `qlm_coded`, or `as_qlm_coded` objects containing predictions to validate. Must include a `.id` column and the variable(s) specified in `by`. Plain data frames are automatically converted to `as_qlm_coded` objects. Multiple objects will be validated separately against the same gold standard, and results combined with a `rater` column to distinguish them.
`gold`	A data frame, `qlm_coded`, or object created with `as_qlm_coded()` containing gold standard annotations. Must include a `.id` column for joining with objects in `...` and the variable(s) specified in `by`. Plain data frames are automatically converted. **Optional** when using objects marked with `as_qlm_coded(data, is_gold = TRUE)` - these are auto-detected.
`by`	Optional. Name of the variable(s) to validate (supports both quoted and unquoted). If `NULL` (default), all coded variables are validated. Can be a single variable (`by = sentiment`), a character vector (`by = c("sentiment", "rating")`), or NULL to process all variables.
`level`	Optional. Measurement level(s) for the variable(s). Can be: • `NULL` (default): Auto-detect from codebook

- Character scalar: Use same level for all variables
- Named list: Specify level for each variable

Valid levels are `"nominal"`, `"ordinal"`, or `"interval"`.

`average` Character scalar. Averaging method for multiclass metrics (nominal level only):

`"macro"` Unweighted mean across classes (default)

`"micro"` Aggregate contributions globally (sum TP, FP, FN)

`"weighted"` Weighted mean by class prevalence

`"none"` Return per-class metrics in addition to global metrics

`ci` Confidence interval method:

`"none"` No confidence intervals (default)

`"analytic"` Analytic CIs where available (ICC, Pearson's r)

`"bootstrap"` Bootstrap CIs for all metrics via resampling

`bootstrap_n` Number of bootstrap resamples when `ci = "bootstrap"`. Default is 1000. Ignored when `ci` is `"none"` or `"analytic"`.

Details

The function performs an inner join between `x` and `gold` using the `.id` column, so only units present in both datasets are included in validation. Missing values (NA) in either predictions or gold standard are excluded with a warning.

Measurement levels:

- **Nominal**: Categories with no inherent ordering (e.g., topics, sentiment polarity). Metrics: accuracy, precision, recall, F1-score, Cohen's kappa (unweighted).
- **Ordinal**: Categories with meaningful ordering but unequal intervals (e.g., ratings 1-5, Likert scales). Metrics: Spearman's rho (`rho`, rank correlation), Kendall's tau (`tau`, rank correlation), and MAE (`mae`, mean absolute error). These measures account for the ordering of categories without assuming equal intervals.
- **Interval/Ratio**: Numeric data with equal intervals (e.g., counts, continuous measurements). Metrics: ICC (intraclass correlation), Pearson's r (linear correlation), MAE (mean absolute error), and RMSE (root mean squared error).

At ordinal and interval level the predictions and the gold standard are read as numbers, whatever type they are stored as, so a column of digits held as text is compared on its values and not on the alphabetical order of its strings. A value that does not read as a number is an error at interval level. Ordinal categories may instead all be text, in which case both columns must be ordered factors with the same levels in the same order, and the categories are ranked by those levels; alphabetical order is never used. A plain factor or a character column at ordinal level is an error that says how to declare the order: `factor(x, levels = c(...), ordered = TRUE)` for data built by hand, or a `type_enum()` written in scale order and declared `"ordinal"` in the codebook, which `qlm_code()` and `as_qlm_coded()` store as that ordered factor.

For multiclass problems with nominal data, the `average` parameter controls how per-class metrics are aggregated:

- **Macro averaging** computes metrics for each class independently and takes the unweighted mean. This treats all classes equally regardless of size.

- **Micro averaging** aggregates all true positives, false positives, and false negatives globally before computing metrics. This weights classes by their prevalence.
- **Weighted averaging** computes metrics for each class and takes the mean weighted by class size.
- **No averaging** (`average = "none"`) returns global macro-averaged metrics plus per-class breakdown.

Note: The `average` parameter only affects precision, recall, and F1 for nominal data. For ordinal data, these metrics are not computed.

Value

A `qlm_validation` object (a tibble/data frame) with the following columns:

`variable` Name of the validated variable

`level` Measurement level used

`measure` Name of the validation metric

`value` Computed value of the metric

`class` For nominal data: averaging method used (e.g., "macro", "micro", "weighted") or class label (when `average = "none"`). For ordinal/interval data: NA (averaging not applicable).

`rater` Name of the object being validated (from input names)

`ci_lower` Lower bound of confidence interval (only if `ci != "none"`)

`ci_upper` Upper bound of confidence interval (only if `ci != "none"`)

The object has class `c("qlm_validation", "tbl_df", "tbl", "data.frame")` and attributes containing metadata (`n`, `call`).

Metrics computed by measurement level:

- **Nominal:** accuracy, precision, recall, f1, kappa
- **Ordinal:** rho (Spearman's), tau (Kendall's), mae
- **Interval:** icc, r (Pearson's), mae, rmse

Confidence intervals:

- `ci = "analytic"`: Provides analytic CIs for ICC and Pearson's r only
- `ci = "bootstrap"`: Provides bootstrap CIs for all metrics via resampling

References

Precision, recall, and F-measure (confusion-matrix definitions and micro / macro averaging): Sokolova, M., & Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4), 427-437. doi:10.1016/j.ipm.2009.03.002

Macro F-measure as the arithmetic mean of per-class F-scores (the convention used here, matching yardstick and scikit-learn): Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval*, Chapter 13. Cambridge University Press. Free online: https://nlp.stanford.edu/IR-book/

Cohen's kappa: Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1), 37-46. doi:10.1177/001316446002000104

Intraclass correlation coefficient: Shrout, P. E., & Fleiss, J. L. (1979). Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*, 86(2), 420-428. doi:10.1037/0033-2909.86.2.420

McGraw, K. O., & Wong, S. P. (1996). Forming inferences about some intraclass correlation coefficients. *Psychological Methods*, 1(1), 30-46. doi:10.1037/1082989X.1.1.30

See Also

Related workflow functions: `qlm_compare()` for inter-rater reliability between coded objects, `qlm_code()` for LLM coding, `as_qlm_coded()` for converting human-coded data.

Underlying classification metrics (internal): `metric_precision()`, `metric_recall()`, `metric_f_meas()`; Cohen's kappa is computed via `reliability_kappa()` and the ICC via `reliability_icc()`.

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# Validate against gold standard (auto-detected)
validation <- qlm_validate(
  examples$example_coded_mini,
  examples$example_gold_standard,
  by = "sentiment",
  level = "nominal"
)
print(validation)

# Explicit gold parameter (backward compatible)
validation2 <- qlm_validate(
  examples$example_coded_mini,
  gold = examples$example_gold_standard,
  by = "sentiment",
  level = "nominal"
)
print(validation2)
```

Index

∗ **datasets**
data_corpus_MPexamples, 14
∗ **data**
data_codebook_immigration, 9
data_codebook_sentiment, 11
data_corpus_LMRDsample, 12
data_corpus_manifsentsUK2010sample, 13
data_corpus_MPexamples, 14
data_corpus_ms2020sample, 16

accessors, 2, *36*, *38*
as_qlm_coded, 4
as_qlm_coded(), *3*, *9*, *14, 15*, *30*, *32–35*, *55, 56*, *58*

codebook(), *3*, *38*
corpus, *13–16*
corpus.qlm_coded, 8
corpus.qlm_translation *(*corpus.qlm_coded*)*, 8
corpus.qlm_translation(), *54*

data_codebook_immigration, 9
data_codebook_sentiment, 11, *12*, *31*
data_corpus_LMRDsample, *11*, 12
data_corpus_manifsentsUK2010sample, *9, 10*, 13
data_corpus_MPexamples, 14
data_corpus_MPexamplesseg *(*data_corpus_MPexamples*)*, 14
data_corpus_ms2020sample, 16

ellmer::batch_chat_structured(), *20*, *22*, *27*
ellmer::chat(), *20*, *22, 23*, *40, 41*, *43*, *51, 52*
ellmer::claude_tool_web_search(), *20*
ellmer::content_image_file(), *30*
ellmer::content_image_url(), *22*, *30*
ellmer::google_tool_web_search(), *20*
ellmer::openai_tool_web_search(), *20*
ellmer::parallel_chat(), *21*, *48*
ellmer::parallel_chat_structured(), *20–22*, *43*
ellmer::params(), *23*, *51*
ellmer::tool(), *20*

inputs(), *3*, *38*, *54*

metric_f_meas(), *58*
metric_precision(), *58*
metric_recall(), *58*

qlm_backfill, 17
qlm_backfill(), *21*, *24–27*, *36*, *42*, *47*, *50*
qlm_code, 19
qlm_code(), *3*, *7*, *10, 11*, *17, 18*, *29–31*, *33*, *35, 36*, *39–44*, *46–48*, *50–52*, *56*, *58*
qlm_codebook, 29
qlm_codebook(), *6*, *10, 11*, *20*, *22*, *27*, *43, 44*
qlm_compare, 31
qlm_compare(), *6, 7*, *11*, *14, 15*, *27*, *30*, *42*, *46*, *58*
qlm_failures, 36
qlm_failures(), *3*, *17, 18*, *21*, *24*, *27*
qlm_meta, 37
qlm_meta(), *3*
qlm_meta<- *(*qlm_meta*)*, 37
qlm_register_provider, 39
qlm_register_provider(), *20*, *27*, *43*, *49*
qlm_replicate, 40
qlm_replicate(), *17, 18*, *21*, *24–27*, *46, 47*
qlm_segment, 42
qlm_segment(), *14, 15*, *32*, *34, 35*, *39*
qlm_trail, 45
qlm_trail(), *3*, *6, 7*, *17, 18*, *20, 21*, *25, 26*, *30*, *47*, *50*, *52*
qlm_transcribe, 47
qlm_transcribe(), *25, 26*, *51, 52*
qlm_translate, 50

qlm_translate(), *9*
qlm_validate, *55*
qlm_validate(), *6, 7*, *27*, *30*, *35*, *46*
quanteda::corpus(), *42, 43*
quanteda::corpus_segment(), *42–44*
quanteda::meta(), *44*

reliability_alpha(), *35*
reliability_alpha_u(), *35*
reliability_icc(), *35*, *58*
reliability_kappa(), *35*, *58*
reliability_kappa_fleiss(), *35*
reliability_kendall_w(), *35*

saveRDS(), *54*

task(), *20*, *30, 31*
type_array(), *29, 30*
type_enum(), *6*, *22*, *29*, *33*, *56*
type_object(), *29, 30*, *43*