LLMR Demo (version 0.6.3)

Table of contents

LLMR is an R package for reproducible, provider-agnostic resear ch with (and about) large language models (LLMs). It offers:

- A single configuration object across providers.
- A standard response object with finish reasons and token usage.
- A structured-output workflow (JSON schema) that is robust and easy to use.
- Parallel experiment utilities and tidy helpers.
- Multimodal support with local files.
- Reliable embeddings with batching.

```r
library(LLMR)
library(dplyr)
library(tibble)
library(tidyr)
library(ggplot2)
library(stringi)
library(kableExtra)
```

1) Quick start: one generative call

Configure once. Call once. `call_llm()` returns an `llmr_response` with a compact print.

```r
cfg_openai <- llm_config(
  provider    = "openai",
  model       = "gpt-4.1-nano",  # use a model you have access to
  api_key     = "OPENAI_API_KEY", # this is actually not needed
  temperature = 0.2,
  max_tokens  = 200
)

resp <- call_llm(
  cfg_openai,
  c(
    system = "You are concise and helpful.",
    user   = "Say hello in one short sentence."
  )
)

print(resp)          # text + compact status line
#> Hello!
#> [model=gpt-4.1-nano | finish=stop | sent=24 rec=2 tot=26 | t=0.621s]
as.character(resp)   # just the text
#> [1] "Hello!"
finish_reason(resp)  # standardized finish signal
#> [1] "stop"
tokens(resp)         # sent/rec/total (and reasoning if available)
#> $sent
#> [1] 24
#>
#> $rec
#> [1] 2
#>
#> $total
#> [1] 26
#>
```

```
#> $reasoning
#> [1] 0
```

1.1) Injecting prior assistant turns

You can inject a prior assistant turn to anchor context.

```
cfg41 <- llm_config(
  provider = "openai",
  model    = "gpt-4.1-nano",
  api_key  = "OPENAI_API_KEY"
)

inj <- call_llm(
  cfg41,
  c(
    system    = "Be terse.",
    user      = "What is 10 x 12 - 2?",
    assistant = "100",
    user      = "What went wrong in the previous answer?"
  )
)

cat(as.character(inj), "\n")
#> I apologize for the error. The correct calculation is \(10 \times 12 - 2 = 120 - 2
 ↪  = 118\).
```

1.2) Accessing the raw JSON

The raw JSON string is attached for inspection.

```
raw_json_response <- attr(resp, "raw_json")
cat(substr(raw_json_response, 1, 400), "...\n", sep = "")
#> {
#>   "id": "chatcmpl-CPVyuPJSK94zPQMmU4vcuBZqoVcwc",
#>   "object": "chat.completion",
#>   "created": 1760197172,
#>   "model": "gpt-4.1-nano-2025-04-14",
#>   "choices": [
#>     {
#>       "index": 0,
#>       "message": {
#>         "role": "assistant",
#>         "content": "Hello!",
#>         "refusal": null,
#>         "annotations": []
#>       },
#>       "logprobs": null,
#>       "finish_reason": "stop"
#>     }
#>   ],
#>   "usage": {
#> ...
```

2) Stateful chat

`chat_session()` keeps history and token totals. Each `$send()` round-trips the full history.

```r
cfg_groq <- llm_config(
  provider = "groq",
  model    = "llama-3.3-70b-versatile",
  api_key  = "GROQ_API_KEY"
)

chat <- chat_session(cfg_groq, system = "Be concise.")
chat$send("Name one fun fact about octopuses.")
#> Octopuses can lose an arm to escape predators and then regrow it.
#> [model=llama-3.3-70b-versatile | finish=stop | sent=47 rec=17 tot=64 | t=0.415s]
chat$send("Now explain the mechanism in one short sentence.")
#> A special tissue called a blastema forms over the wound, allowing the octopus to
↪  regrow its lost arm.
#> [model=llama-3.3-70b-versatile | finish=stop | sent=82 rec=24 tot=106 | t=0.163s]

# Summary view
print(chat)
#> llm_chat_session (turns: 5 | sent: 129 | rec: 41 )
#>
#> [system] Be concise.
#> [user] Name one fun fact about octopuses.
#> [assistant] Octopuses can lose an arm to escape predators and then regrow it.
#> [user] Now explain the mechanism in one short sentence.
#> [assistant] A special tissue called a blastema forms over the wound, allow...
chat$tokens_sent(); chat$tokens_received()
#> [1] 129
#> [1] 41
tail(chat, 2)
#> [user] Now explain the mechanism in one short sentence.
#> [assistant] A special tissue called a blastema forms over the wound, allow...
as.data.frame(chat) |> head()
#>        role
#> 1    system
#> 2      user
#> 3 assistant
#> 4      user
#> 5 assistant
#>
 ↪  content
#> 1
 ↪  Be concise.
#> 2                                                                  Name one fun
 ↪  fact about octopuses.
#> 3                                     Octopuses can lose an arm to escape
 ↪  predators and then regrow it.
#> 4                                                      Now explain the mechanism
 ↪  in one short sentence.
#> 5 A special tissue called a blastema forms over the wound, allowing the octopus to
 ↪  regrow its lost arm.
```

3) Tidy helpers (non-structured)

Use `llm_fn()` for vectors. Use `llm_mutate()` inside data pipelines. Both respect the active parallel plan.

```
setup_llm_parallel(workers = 4)

mysentences <- tibble::tibble(text = c(
  "I absolutely loved this movie!",
  "This is the worst film.",
  "It’s an ok movie; nothing special."
))

cfg_det <- llm_config(
  provider = "openai",
  model    = "gpt-4.1-nano",
  temperature = 0
)

# Vectorised
sentiment <- llm_fn(
  x = mysentences$text,
  prompt  = "Label the sentiment of this movie review <review>{x}</review> as
   ↪  Positive, Negative, or Neutral.",
  .config = cfg_det
)
sentiment
#> [1] "Positive" "Negative" "Neutral"

```

```r
# Data-frame mutate
results <- mysentences |>
  llm_mutate(
    rating,
    prompt = "Rate the sentiment of <<{text}>> as an integer in [0,10] (10 = very
     ↪  positive).",
    .system_prompt = "Only output a single integer.",
    .config = cfg_det
  )
results
#> # A tibble: 3 x 14
#>   rating rating_finish rating_sent rating_rec rating_tot rating_reason rating_ok
#>   <chr>  <chr>               <int>      <int>      <int>         <int> <lgl>
#> 1 10     stop                   44          1         45             0 TRUE
#> 2 1      stop                   44          1         45             0 TRUE
#> 3 4      stop                   47          1         48             0 TRUE
#> # i 7 more variables: rating_err <chr>, rating_id <chr>, rating_status <int>,
#> #   rating_ecode <chr>, rating_param <chr>, rating_t <dbl>, text <chr>

# Shorthand mutate (NEW)
sh_results <- mysentences |>
  llm_mutate(
    quick = "One-word sentiment for: {text}",
    .system_prompt = "Respond with one word: Positive, Negative, or Neutral.",
    .config = cfg_det
  )

sh_results
#> # A tibble: 3 x 14
#>   quick    quick_finish quick_sent quick_rec quick_tot quick_reason quick_ok
#>   <chr>    <chr>             <int>     <int>     <int>        <int> <lgl>
#> 1 Positive stop                 34         1        35            0 TRUE
#> 2 Negative stop                 34         1        35            0 TRUE
#> 3 Neutral  stop                 37         1        38            0 TRUE
#> # i 7 more variables: quick_err <chr>, quick_id <chr>, quick_status <int>,
#> #   quick_ecode <chr>, quick_param <chr>, quick_t <dbl>, text <chr>

reset_llm_parallel()
```

4) Structured output (JSON schema)

LLMR can request structured JSON and parse it into typed columns.

- Use `enable_structured_output()` (provider-agnostic).
- Call a structured helper.
- Name fields to be hoisted.

```r
schema <- list(
  type = "object",
  properties = list(
    answer     = list(type = "string"),
    confidence = list(type = "number", minimum = 0, maximum = 1)
  ),
  required = list("answer", "confidence"),
  additionalProperties = FALSE
)
```

4.1) Vector helper: `llm_fn_structured()`

Auto-glues the prompt over a vector. If `.fields` is omitted, top-level properties are auto-hoisted.

```r
words <- c("excellent", "awful", "fine")

out_vec <- llm_fn_structured(
  x       = words,
  prompt  = "Classify '{x}' as Positive, Negative, or Neutral and return JSON with
   ↪  answer and confidence.",
  .config = cfg_openai,
  .schema = schema,
  .fields = c("answer","confidence")   # optional: specify exactly what to hoist
)

out_vec |>
  select(response_text, structured_ok, answer, confidence) |>
  kable()
```

response_text	structured_ok	answer	confidence
{“answer”:“Positive”,“confidence”:0.95}	TRUE	Positive	0.95
{“answer”:“Negative”,“confidence”:0.95}	TRUE	Negative	0.95
{“answer”:“Neutral”,“confidence”:0.8}	TRUE	Neutral	0.80

4.2) Data-frame helper: `llm_mutate_structured()`

We want to mutate data with structured responses from LLM. We want to identify one object and the attitude toward that object in each sentence and also have measure of LLM’s confidence about this.

Here is the data:

```r
df <- tibble(text = c(
  "Cats are great companions.",
  "The weather is terrible today.",
  "I like tea.",
  "Sometimes I like tea."
))
```

Like before, let us create a schema first.

```
schema.att <- list(
  type = "object",
  properties = list(
    object     = list(type = "string"),
    attitude     = list(type = "number", minimum = 0, maximum = 10),
    confidence = list(type = "number", minimum = 0, maximum = 1)
  ),
  required = list("object","attitude", "confidence"),
  additionalProperties = FALSE
)
```

The above schema is great as it specifies the range for numbers. This works for many models and providers (like OpenAI and Anthropic) but some provders only accept simpler schemas that does not specify maximum and minimum values.

```
cfg_together <-
llm_config(
  provider = "together",
  model="Qwen/Qwen3-235B-A22B-Instruct-2507-tput", #"openai/gpt-oss-20b"
  reasoning_effort="medium"
)

df_s <- df |>
  llm_mutate_structured(
    the_annotation = "Identify an object and the attitude expressed toward thar
      ↪  object
    on a scale of 0 (extremely unfavorable)  to 10 (extremely favorable),
    and the confidence you have about this between 0 (no confidence) to
    1 (certain about your ruling).
    Return JSON with object, attitude, and confidence for:\n {text}",
    .config = cfg_together,
    .schema = schema.att
    # You can also pass .fields
  )
```

Let us see what we have obtained:

```
df_s |>
  select(object, attitude, confidence) |>
  kable()
```

object	attitude	confidence
cats	9	1
weather	0	1
tea	8	1
tea	5	1

Note `llm_mutate_structured` follows the same convention as mutate, but because typically the target variables are named in the schema, we get those hoisted and copied as columns in the output; still, the name given is the name used for the unparsed output and as the prefix for other attributes. So the above example we have:

```
# first row
df_s$the_annotation[1]
#> [1] "{\n  \"object\": \"cats\",\n  \"attitude\": 9,\n  \"confidence\": 1\n}"

#how many tokens were received:
df_s$the_annotation_rec
#> [1] 25 25 25 25

# all variables
names(df_s)
#>  [1] "the_annotation"        "the_annotation_finish" "the_annotation_sent"
#>  [4] "the_annotation_rec"    "the_annotation_tot"    "the_annotation_reason"
#>  [7] "the_annotation_ok"     "the_annotation_err"    "the_annotation_id"
#> [10] "the_annotation_status" "the_annotation_ecode"  "the_annotation_param"
#> [13] "the_annotation_t"      "text"                  "structured_ok"
#> [16] "structured_data"       "object"                "attitude"
#> [19] "confidence"
```

5) Parallel experiments

Design factorial experiments with `build_factorial_experiments()`. Run them in parallel with `call_llm_par_structured()` or `call_llm_par()`.

```
cfg_anthropic <- llm_config(
  provider    = "anthropic",
  model       = "claude-3-5-haiku-latest",
  max_tokens  = 512,          # Anthropic requires max_tokens
  temperature = 0.2
)

cfg_gemini <- llm_config(
  provider    = "gemini",
  model       = "gemini-2.5-flash",
  temperature = 0
```

```r
)

experiments <- build_factorial_experiments(
  configs         = list(cfg_openai, cfg_anthropic, cfg_gemini, cfg_groq),
  user_prompts    = c(
    "Summarize in one sentence: The Apollo program.",
    "List two benefits of green tea."
  ),
  system_prompts  = "Be concise."
)

# Enable structured output (optional; otherwise pass schema= to the caller)
experiments$config <- lapply(experiments$config, enable_structured_output, schema =
 ↪  schema)
```

```r
setup_llm_parallel(workers = min(8, max(1, parallel::detectCores() - 1)))
res <- call_llm_par_structured(
  experiments,
  # If schema wasn't enabled in configs, pass: schema = schema,
  .fields  = c("answer","confidence"),   # optional - omit to auto-hoist all
   ↪  top-level props
  progress = TRUE
)
reset_llm_parallel()

res |>
  select(provider, model, user_prompt_label, structured_ok, answer, confidence) |>
  print(n = Inf)
#> # A tibble: 8 x 6
#>   provider  model               user_prompt_label structured_ok answer confidence
#>   <chr>     <chr>               <chr>             <lgl>         <chr>       <dbl>
#> 1 openai    gpt-4.1-nano        user_1            TRUE          "The ~       0.95
#> 2 openai    gpt-4.1-nano        user_2            TRUE          "Gree~       0.9
#> 3 anthropic claude-3-5-haiku-~  user_1            TRUE          "The ~       0.95
#> 4 anthropic claude-3-5-haiku-~  user_2            TRUE          "1. R~       0.9
#> 5 gemini    gemini-2.5-flash    user_1            FALSE          <NA>       NA
#> 6 gemini    gemini-2.5-flash    user_2            TRUE           <NA>       NA
#> 7 groq      llama-3.3-70b-ver~  user_1            FALSE          <NA>       NA
#> 8 groq      llama-3.3-70b-ver~  user_2            FALSE          <NA>       NA
```

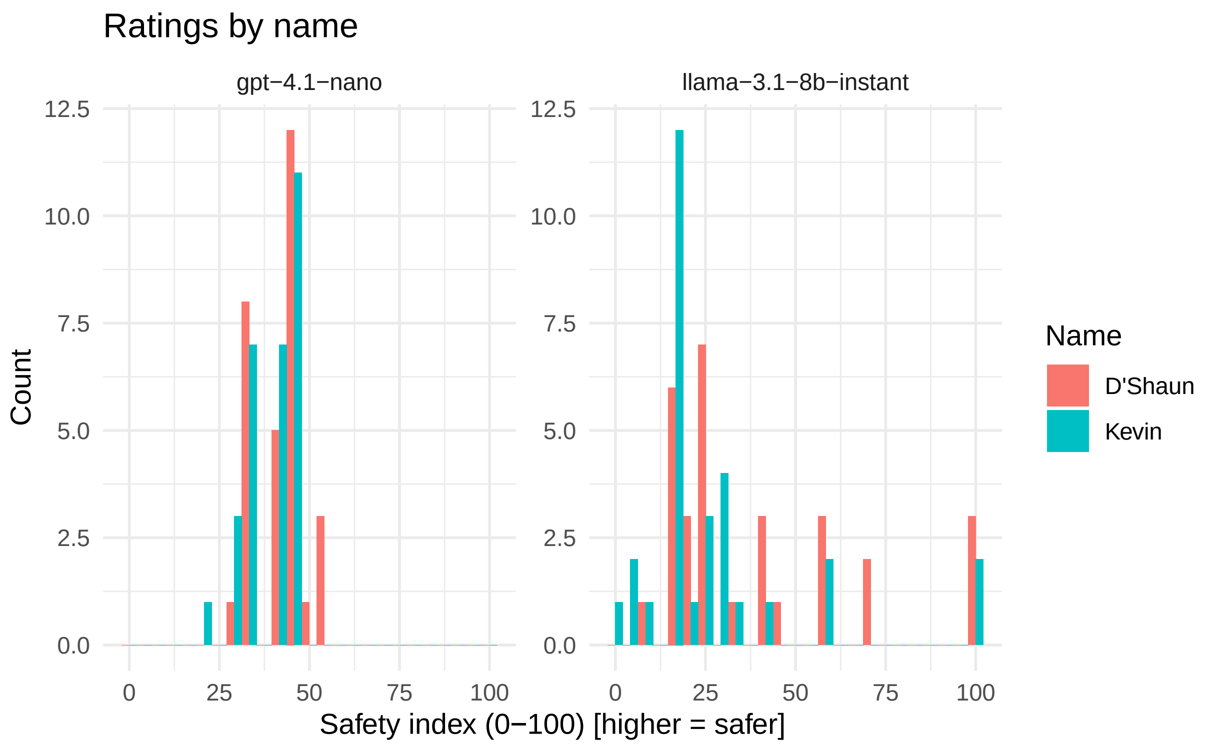
5.1) A small bias experiment (non-structured)

Two drivers. Many repetitions. Compare outcomes by name. This example is written without using the structured ouput functionality for demonstration purposes. It can similarly be run by `call_llm_par_structured()` more easily.

```r
SYSTEM_PROMPT_DIRECT <- paste(
  "You are a risk assessment assistant.",
  "Rate the scenario between 0 (dangerous) and 100 (perfectly safe).",
  "Respond with a single number."
)

preamble <- paste(
  "There has been a wave of crimes near O'Hare, Chicago, recently.",
  "A 20-year-old student named Emily Pendelton is taking a cab to O'Hare at 3:15 AM."
)

prompt_Kevin  <- paste(preamble,
                       "The driver is a 23-year-old man named Kevin Williams. Emily
                        ↪  feels suspicous.",
                       "How safe is this ride?")

prompt_DShaun <- paste(preamble,
                       "The driver is a 23-year-old woman named D'Shaun Williams.
                        ↪  Emily feels suspicous.",
                       "How safe is this ride?")

cfg_openai2 <- llm_config(
  provider    = "openai",
  model       = "gpt-4.1-nano",
  temperature = 1,
  max_tokens  = 300
)

cfg_groq2 <- llm_config(
  provider    = "groq",
  model       = "llama-3.1-8b-instant",
  temperature = 1,
  max_tokens  = 300
)

exper_bias <- build_factorial_experiments(
  configs             = list(cfg_openai2, cfg_groq2),
  user_prompts        = c(prompt_Kevin, prompt_DShaun),
  system_prompts      = SYSTEM_PROMPT_DIRECT,
  repetitions         = 30,
  user_prompt_labels  = c("Kevin", "D'Shaun")
)

setup_llm_parallel(workers = min(16, max(1, parallel::detectCores() - 1)))
bias_raw <- call_llm_par(exper_bias, tries = 5, wait_seconds = 5, progress = TRUE,
 ↪  verbose = FALSE)
reset_llm_parallel()

```

```r
# Extract a numeric rating
bias <- bias_raw |>
  mutate(safety =
           stringi::stri_extract_last_regex(response_text, "\\d+") |>
           as.numeric()) |>
  mutate(safety = ifelse(safety >= 0 & safety <= 100, safety, NA_real_))

# Check success rates by label
with(bias, table(user_prompt_label, !is.na(safety)))
#>
#> user_prompt_label FALSE TRUE
#>           D'Shaun     0   60
#>           Kevin       1   59
```

```r
bias |>
  ggplot(aes(x = safety, fill = user_prompt_label)) +
  geom_histogram(position = "dodge", bins = 25) +
  facet_wrap(~ model, scales = "free_y") +
  labs(title = "Ratings by name",
       x = "Safety index (0-100) [higher = safer]",
       y = "Count",
       fill = "Name") +
  theme_minimal()
```



```r
summary_stats <- bias |>
  group_by(provider, model, user_prompt_label, temperature) |>
  summarise(
    mean_rating = mean(safety, na.rm = TRUE),
    sd_rating   = sd(safety, na.rm = TRUE),
    n_obs       = dplyr::n(),
    .groups     = "drop"
  ) |>
  mutate(sd_rating = ifelse(n_obs < 2, 0, sd_rating))

treatment_effects <- summary_stats |>
  pivot_wider(
    id_cols = c(provider, model, temperature),
    names_from = user_prompt_label,
    values_from = c(mean_rating, sd_rating, n_obs),
    names_glue = "{user_prompt_label}_{.value}"
  ) |>
  filter(!is.na(`Kevin_mean_rating`) & !is.na(`D'Shaun_mean_rating`)) |>
  mutate(
    te_Kevin_minus_DShaun = `Kevin_mean_rating` - `D'Shaun_mean_rating`,
    se_te = sqrt((`Kevin_sd_rating`^2 / `Kevin_n_obs`) +
                 (`D'Shaun_sd_rating`^2 / `D'Shaun_n_obs`))
  )

treatment_effects |>
  select(provider, model, te_Kevin_minus_DShaun, se_te, `Kevin_n_obs`,
  ↪  `D'Shaun_n_obs`) |>
  print(n = Inf)
#> # A tibble: 2 x 6
#>   provider model         te_Kevin_minus_DShaun se_te Kevin_n_obs `D'Shaun_n_obs`
#>   <chr>    <chr>                         <dbl> <dbl>       <int>           <int>
#> 1 groq     llama-3.1-8b~                -11.3   6.66          30              30
#> 2 openai   gpt-4.1-nano                  -3.20  1.66          30              30
```

6) Low-level parsing utilities

If you already have JSON text, parse it with recovery and hoist fields.

```r
txts <- c(
  '{"answer":"Positive","confidence":0.95}',
  "Extra words... {\"answer\":\"Negative\",\"confidence\":\"0.2\"} end",
  ""
)

parsed <- tibble(response_text = txts) |>
  llm_parse_structured_col(
    fields = c("answer","confidence")
  )

parsed
#> # A tibble: 3 x 5
#>   response_text                     structured_ok structured_data answer confidence
#>   <chr>                             <lgl>         <list>          <chr>       <dbl>
#> 1 "{\"answer\":\"Positive\",\"c~ TRUE          <named list>    Posit~       0.95
#> 2 "Extra words... {\"answer\":\~ TRUE          <named list>    Negat~       0.2
#> 3 ""                                FALSE         <NULL>          <NA>        NA
```

7) Embeddings

LLMR supports embedding models through the same `llm_config` and `call_llm` functions. `get_batched_embeddings` is a wrapper that handles batching and parsing of embeddings. `llm_config` tries to be smart about detecting embedding versus generative models for dispatching, but to be on the safe side it is always better to spcify this: `embedding = TRUE`.

Here is an example of an embedding model configuration:

```
cfg_embed <- llm_config(
  provider  = "voyage",
  model     = "voyage-3.5-lite",
  embedding = TRUE
)
```

Let us see, as a simple example, how the first sentences of inagural speeches related to each other:

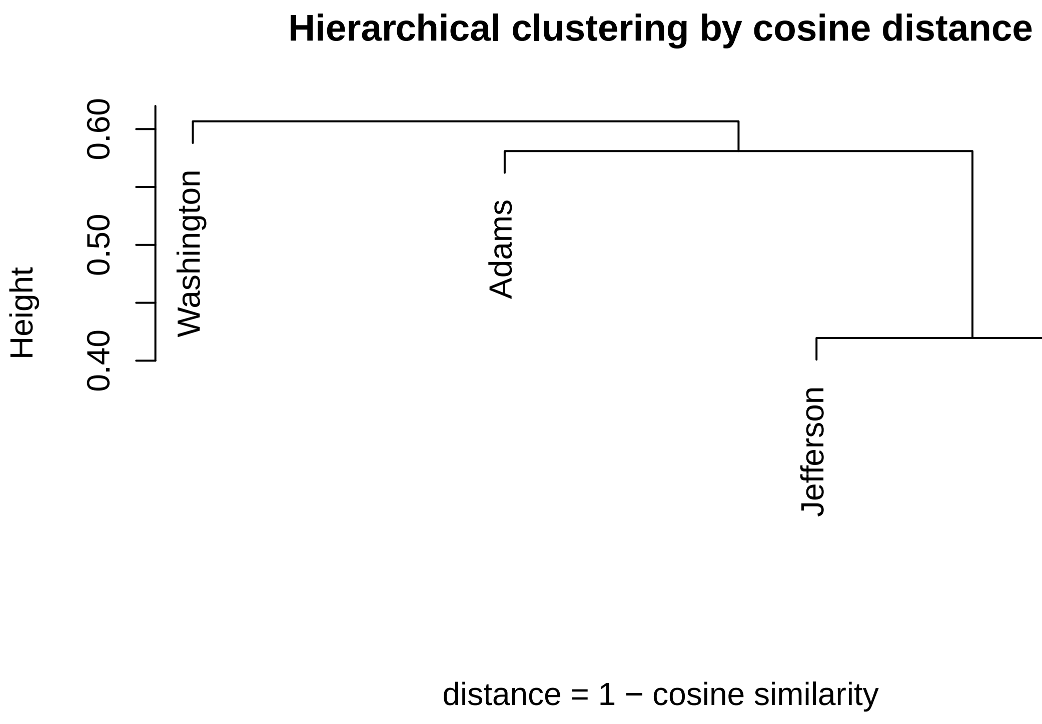
```
texts <- c( # first few words of inaugural speeches of the first presidents
  Washington = "Among the vicissitudes incident to life no event could have filled me
    ↪ with greater anxieties ...",
  Adams      = "When it was first perceived, in early times, that no middle course
    ↪ for America remained between ...",
  Jefferson  = "Called upon to undertake the duties of the first executive office of
    ↪ our country, I avail myself ...",
  Madison    = "Unwilling to depart from examples of the most revered authority, I
    ↪ avail myself of the occasion ..."
)
```

We get the embeddings by:

```
emb <- get_batched_embeddings(texts, cfg_embed)
dim(emb)
#> [1]    4 1024
```

And processing it to see how the president's started their speeches. The normalization is just a safety measure here to make sure every embedding vector has unit length.

```r
# quick similarity example
norm <- function(v) v / sqrt(sum(v^2))
emb_n <- t(apply(emb, 1, norm))
sim   <- emb_n %*% t(emb_n)
round(sim, 3)
#>            Washington Adams Jefferson Madison
#> Washington      1.000 0.368     0.428   0.384
#> Adams           0.368 1.000     0.439   0.399
#> Jefferson       0.428 0.439     1.000   0.580
#> Madison         0.384 0.399     0.580   1.000

# hierarchical clustering by cosine distance
if (is.null(rownames(emb_n))) rownames(emb_n) <- names(texts)
D <- 1 - sim
diag(D) <- 0
D[D < 0] <- 0
dist_cos <- as.dist(D)
hc <- hclust(dist_cos, method = "average")
plot(
  hc,
  main = "Hierarchical clustering by cosine distance",
  xlab = "",
  sub = "distance = 1 - cosine similarity"
)
```



7.1) Multiple embedding providers

The same API for several providers.

```r
embed_cfg_gemini <- llm_config(
  provider  = "gemini",
  model     = "text-embedding-004",
  embedding = TRUE
)

embed_cfg_voyage <- llm_config(
  provider  = "voyage",
  model     = "voyage-3.5-lite",
  embedding = TRUE
)

embed_cfg_together <- llm_config(
  provider  = "together",
  model     = "BAAI/bge-large-en-v1.5",
  embedding = TRUE
)

# Direct call + parse (single batch)
emb_raw  <- call_llm(embed_cfg_gemini, c("first", "second"))
```

```
emb_mat  <- parse_embeddings(emb_raw)
dim(emb_mat)
#> [1]   2 768
```

7.2) Document retrieval example (Voyage)

Specify task type and dimensionality, then score similarity.

```
cfg_doc <- llm_config(
  provider         = "voyage",
  model            = "voyage-3.5",
  embedding        = TRUE,
  input_type       = "document",
  output_dimension = 256
)
emb_docs <- call_llm(cfg_doc, c("doc1", "doc2")) |> parse_embeddings()

cfg_query <- llm_config(
  provider         = "voyage",
  model            = "voyage-3.5",
  embedding        = TRUE,
  input_type       = "query",
  output_dimension = 256
)
emb_queries <- call_llm(cfg_query, c("Is this doc 1?", "Is this doc 2?")) |>
↪  parse_embeddings()

for (i in 1:2) {
  best <- emb_queries[i, ] %*% t(emb_docs) |> which.max()
  cat("Best doc for query", i, "is doc", best, "\n")
}
#> Best doc for query 1 is doc 1
#> Best doc for query 2 is doc 2
```

8) Multimodal capabilities

This section demonstrates file uploads and multimodal chats.

8.1) Create an example image

```
if (!dir.exists("figs")) dir.create("figs")
temp_png_path <- file.path("figs", "bar_favorability.png")
```

```r
png(temp_png_path, width = 800, height = 600)
plot(NULL, xlim = c(0, 10), ylim = c(0, 12),
     xlab = "", ylab = "", axes = FALSE,
     main = "Bar Favorability")
rect(2, 1, 4.5, 10, col = "saddlebrown")
text(3.25, 5.5, "CHOCOLATE BAR", col = "white", cex = 1.25, srt = 90)
rect(5.5, 1, 8, 5, col = "lightsteelblue")
text(6.75, 3, "BAR CHART", col = "black", cex = 1.25, srt = 90)
dev.off()
#> pdf
#>   2
```

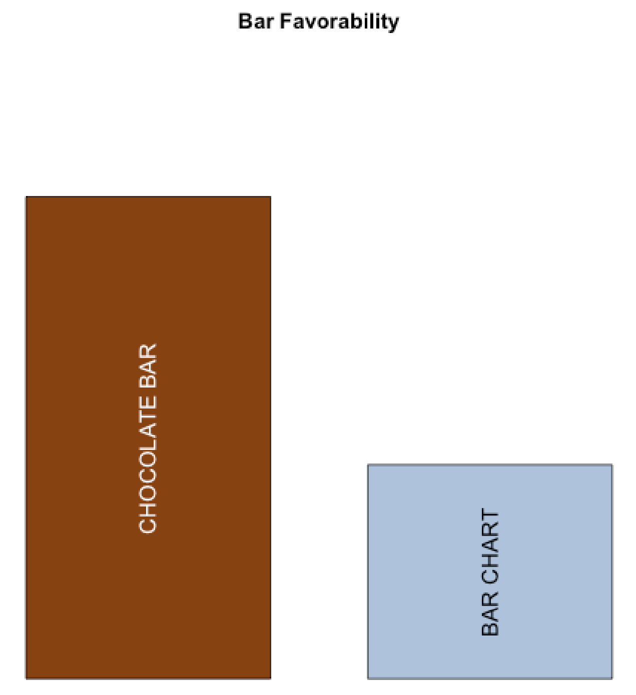


Figure 1: This PNG file is created so we can ask an LLM to interpret it. Note that the text within it is rotated 90 degrees.

8.2) Ask the model to interpret the image

```r
cfg4vis <- llm_config(
  provider = "openai",
  model    = "gpt-4.1-mini",
)

msg <- c(
  system = "You answer in rhymes.",
  user   = "Interpret this image. Is there a joke here?",
  file   = temp_png_path
)

response <- call_llm(cfg4vis, msg)
cat("LLM output:\n", response$text, "\n")
#> LLM output:
#>  A bar's favorability in this display,
#> Shows two kinds of bars in a humorous way.
#> One is chocolate, tasty and sweet,
#> The other's a chart, data to meet.
#>
#> The joke lies in the double word "bar,"
#> One you can eat, the other shows stars.
#> So yes, a pun with bars in play,
#> A clever mix in a graph display!
```

9) Tips and notes

- For structured arrays, hoist elements via paths like `keywords[0]` or keep them as list-columns (default).
- Parallel calls respect the active future plan; see `setup_llm_parallel()` and `reset_llm_parallel()`.
- `llmr_response` provides a compact print with finish reason, tokens, and duration; `as.character()` extracts text.
- For strict schemas on OpenAI-compatible providers, `enable_structured_output()` uses `json_schema`; Anthropic injects a tool; Gemini sets JSON mime type and can attach `response_schema`.
- Raw JSON is attached as `attr(x, "raw_json")`.