"""

LLM Reviewing Framework – generic, best‑practice prompt design

Author: ChatGPT demo – June 10 2025

Overview

========

A lightweight \*\*async\*\* framework that lets you drop any piece of text (e.g. a memo drafted by a Chief‑of‑Staff, a marketing blurb, a policy doc) into a pool of LLM “critics”, aggregate their numeric ratings, and—optionally—loop through revisions until a very high quality bar is met.

Why this is \*framework‑style\*

----------------------------

\* No hard‑coded roles or subject matter.

\* The default critic persona (“Domain Coach”) is intentionally broad: it judges \*\*accuracy, clarity, completeness, strategic relevance, practicality, coherence, and policy compliance\*\*. Swap it for your own rubric by editing a single string.

\* Generation is \*pluggable\*: pass `generator=None` if you only want to review a human‑written draft; supply a generator coroutine if you want auto‑revision.

Key knobs (all overridable)

---------------------------

\* `model` → `gpt‑4o‑mini` (cheap & fast)

\* `pass\_threshold`→ \*\*9.5\*\* (difficult‑high)

\* `max\_rounds` → \*\*7\*\*

\* `reviewers` → three \*\*Domain Coach\*\* clones with different temperatures for diversity.

Fail‑safe wording

----------------

If the critics still haven’t reached 9.5 after the final round, the framework returns the highest‑scoring attempt prefaced with:

> “I do not believe this output is good but for now this will do…”

Set the `OPENAI\_API\_KEY` environment variable or replace `openai\_key\_here` before running.

"""

from \_\_future\_\_ import annotations

import asyncio

import json

import logging

import os

from dataclasses import dataclass

from typing import Any, Callable, Coroutine, List, Optional

import openai

# -------------------------------------------------

# Data classes

# -------------------------------------------------

@dataclass

class ReviewResult:

reviewer: str

score: float

verdict: str

feedback: str

def to\_dict(self) -> dict[str, Any]:

return {

"reviewer": self.reviewer,

"score": self.score,

"verdict": self.verdict,

"feedback": self.feedback,

}

@dataclass

class ReviewerConfig:

name: str

system\_prompt: str

weight: float = 1.0

model: str = "gpt-4o-mini" # fast & affordable

temperature: float = 0.2 # each copy can vary for stochastic diversity

# -------------------------------------------------

# Reviewer agent

# -------------------------------------------------

class Reviewer:

"""A single LLM critic that returns a structured JSON review."""

def \_\_init\_\_(self, config: ReviewerConfig):

self.config = config

async def review(self, context: str, answer: str) -> ReviewResult:

"""Ask the LLM to grade \*answer\* versus \*context\* (could be an empty string)."""

rubric\_query = (

"You are {name}. You will receive some CONTEXT (the task or brief) and an ANSWER (a draft response).\n\n"

"Evaluate the ANSWER on these axes: accuracy, completeness, clarity, coherence, strategic relevance, practicality, and policy compliance.\n"

"Return a score from 1 (poor) to 10 (excellent).\n"

"If score < 9.5 provide \*\*one concise, actionable improvement suggestion\*\*.\n"

"Respond \*\*strictly\*\* in JSON \*only\* with the keys: score, verdict (pass/partial/fail), feedback."

).format(name=self.config.name)

resp = await openai.ChatCompletion.acreate(

model=self.config.model,

temperature=self.config.temperature,

messages=[

{"role": "system", "content": self.config.system\_prompt},

{"role": "user", "content": f"CONTEXT:\n{context}\n\nANSWER:\n{answer}\n\n{rubric\_query}"},

],

)

content = resp.choices[0].message.content.strip()

try:

data = json.loads(content)

except json.JSONDecodeError:

logging.warning("%s returned non‑JSON: %s", self.config.name, content)

data = {"score": 0, "verdict": "invalid", "feedback": "Could not parse JSON"}

return ReviewResult(

reviewer=self.config.name,

score=float(data.get("score", 0)),

verdict=str(data.get("verdict", "invalid")),

feedback=str(data.get("feedback", "")),

)

# -------------------------------------------------

# Review manager / orchestrator

# -------------------------------------------------

class ReviewManager:

"""Coordinates reviewers, aggregates scores, and (optionally) loops through generations."""

def \_\_init\_\_(

self,

reviewers: List[ReviewerConfig],

pass\_threshold: float = 9.5,

max\_rounds: int = 7,

):

self.reviewer\_agents = [Reviewer(cfg) for cfg in reviewers]

self.configs = reviewers

self.pass\_threshold = pass\_threshold

self.max\_rounds = max\_rounds

async def review\_only(self, context: str, answer: str) -> dict[str, Any]:

"""One‑shot review with no iterative improvement."""

reviews = await self.\_gather\_reviews(context, answer)

avg\_score = self.\_weighted\_average(reviews)

return {

"average\_score": avg\_score,

"reviews": [r.to\_dict() for r in reviews],

"passed": avg\_score >= self.pass\_threshold,

}

async def evaluate(

self,

context: str,

initial\_answer: str,

generator: Optional[Callable[[str, str, int], Coroutine[Any, Any, str]]] = None,

) -> dict[str, Any]:

"""Review the initial answer, then optionally iterate using \*generator\*.

\*context\* → the task / brief / background (can be empty).

\*initial\_answer\* → draft to evaluate (from the Chief‑of‑Staff, etc.).

\*generator\* → async func(context, prev\_answer, round\_idx) -> new\_answer.

Pass None to skip iterative improvement.

"""

outcomes: List[dict[str, Any]] = []

best\_answer = initial\_answer

best\_score, passed = -1.0, False

current\_answer = initial\_answer

for round\_idx in range(1, (self.max\_rounds if generator else 1) + 1):

reviews = await self.\_gather\_reviews(context, current\_answer)

avg\_score = self.\_weighted\_average(reviews)

outcomes.append(

{

"round": round\_idx,

"answer": current\_answer,

"average\_score": avg\_score,

"reviews": [r.to\_dict() for r in reviews],

}

)

if avg\_score > best\_score:

best\_score, best\_answer = avg\_score, current\_answer

if avg\_score >= self.pass\_threshold:

passed = True

break

if generator is None:

break # one‑shot review

# Ask generator for a revision based on feedback

current\_answer = await generator(context, current\_answer, round\_idx + 1)

final\_answer = (

best\_answer

if passed

else "I do not believe this output is good but for now this will do…\n\n" + best\_answer

)

return {

"passed": passed,

"best\_score": best\_score,

"final\_answer": final\_answer,

"outcomes": outcomes,

}

# ----------------------- helpers -----------------------

async def \_gather\_reviews(self, context: str, answer: str) -> List[ReviewResult]:

tasks = [agent.review(context, answer) for agent in self.reviewer\_agents]

return await asyncio.gather(\*tasks)

def \_weighted\_average(self, results: List[ReviewResult]) -> float:

total, wsum = 0.0, 0.0

for cfg, res in zip(self.configs, results):

total += cfg.weight \* res.score

wsum += cfg.weight

return total / wsum if wsum else 0.0

# -------------------------------------------------

# Example usage (remove or adapt in production)

# -------------------------------------------------

async def demo():

"""Demonstrates \*review‑only\* mode on a Chief‑of‑Staff draft."""

openai.api\_key = os.getenv("OPENAI\_API\_KEY", "openai\_key\_here")

# Generic, multi‑functional critic prompt

universal\_prompt = (

"You are a seasoned Domain Coach. Your job: critically and constructively evaluate any text for "

"accuracy, completeness, clarity, coherence, strategic relevance, practicality, and policy compliance."

)

reviewer\_cfgs = [

ReviewerConfig(name=f"Domain Coach {c}", system\_prompt=universal\_prompt, temperature=t)

for c, t in zip(["A", "B", "C", "D"], [0.2, 0.4, 0.7, 0.9])

]

manager = ReviewManager(reviewer\_cfgs)

# ---------- Chief‑of‑Staff draft (input to be reviewed) ----------

cos\_draft = (

"Team — Q2 was a whirlwind! \n\nRevenue +34% QoQ driven by our self‑serve funnel. CAC held flat at $112, "

"but churn upticked to 5.1%. Biggest win: we shipped the AI‑assist beta to 200 logos. Biggest challenge: "

"data infra latency causing 6 P0 incidents. Ask: an extra $400k to fast‑track the ingestion pipeline refactor. "

"Full break‑down in Slide #7."

)

context = "Internal investor update memo, <200 words." # optional

results = await manager.evaluate(context, cos\_draft, generator=None)

# Print quick summary

print("---- PASSED?", results["passed"], "SCORE", results["best\_score"])

print("---- FINAL ANSWER ----\n", results["final\_answer"])

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(demo())