```
# AI_assistant.py
import time
import queue
import threading
import datetime
import re
import ollama
import textwrap

from speech import speech
from memory import memory
from listen import listen
from Repeat_Last import repeat
from GUI import gui

message_no_date = ""

from en_to_af import translate

him_her = ""
madam_sir = ""


def extract_message(text: str) -> str:
    """
    Removes leading timestamp (date + time) from a query string.
    Example:
        '2025-09-04 : 21:28:43 : players i've learned it'
        -> "players i've learned it"
    """
    # Match: YYYY-MM-DD : HH:MM:SS : message
    match = re.match(r"^\d{4}-\d{2}-
\d{2}\s*:\s*\d{2}:\d{2}:\d{2}\s*:\s*(.*)", text)
    return match.group(1).strip() if match else text.strip()


def format_ollama_response(raw: str, wrap_width: int = 80) -> str:
    """
    Convert a messy Ollama response into readable Markdown:
      - preserves fenced code blocks (``` ... ```)
      - converts **Bold** headings into Markdown headings (##) for non-
code parts
      - wraps normal paragraphs to wrap_width
      - preserves and wraps list items
      - heuristic: if no fences are present but text *looks like* code,
wrap it in ```python
    """
    if not raw:
        return ""

    # normalize line endings but keep leading/trailing whitespace for now
    text = raw.replace('\r\n', '\n')
```

```
    # Heuristic: if no explicit fences exist but text looks code-like,
wrap it
    if '```' not in text:
        code_like_lines = sum(
            1 for ln in text.splitlines()
            if ln.strip().startswith(('import ', 'def ', 'class ', 'from
', 'cv2', 'np', '#'))
        )
        if code_like_lines >= 3:
            text = "```python\n" + text.strip() + "\n```"

    # Use a robust regex to split code blocks and keep them intact
    CODE_BLOCK_RE = re.compile(r'(```[\s\S]*?```)', re.DOTALL)
    parts = CODE_BLOCK_RE.split(text)

    # Heading regex we will APPLY ONLY to non-code parts
    HEADING_BOLD_RE = re.compile(r'^\s*\*{2}(.+?)\*{2}\s*$',
flags=re.MULTILINE)

    out_chunks = []
    for part in parts:
        if not part:
            continue

        part_stripped = part.strip()

        # If this is a fenced code block, preserve it unchanged (don't
reflow)
        if part_stripped.startswith('```') and
part_stripped.endswith('```'):
            # preserve exactly but ensure one trailing newline for
separation
            # don't call .strip() on the inner content (preserves
indentation)
            # normalize to have a single trailing newline
            block = part_stripped + '\n'
            out_chunks.append(block)
            continue

        # Non-code chunk: apply heading conversion then paragraph
wrapping
        non_code = HEADING_BOLD_RE.sub(r'## \1', part)

        # Insert blank lines before common section starters for
readability
        for key in ["Example Usage", "Requirements", "Key technical
notes",
                    "Edge cases & robustness", "Suggestions to extend
functionality"]:
            non_code = re.sub(r'\n\s*(' + re.escape(key) + r')',
r'\n\n\1', non_code, flags=re.IGNORECASE)

        # collapse excessive blank lines, trim edges
```

```
        chunk = re.sub(r'\n\s*\n+', '\n\n', non_code).strip()
        if not chunk:
            continue

        paragraphs = chunk.split('\n\n')
        formatted_pars = []
        for para in paragraphs:
            lines = [ln.rstrip() for ln in para.splitlines() if
ln.strip() != '']
            if not lines:
                continue
            # single-line heading
            if len(lines) == 1 and lines[0].strip().startswith('##'):
                formatted_pars.append(lines[0].strip())
                continue

            # Detect list items (bullets or numbered)
            if all(re.match(r'^\s*([-*•]|\d+\.)\s+', ln) for ln in
lines):
                wrapped_items = []
                for ln in lines:
                    m = re.match(r'^(\s*([-*•]|\d+\.)\s+)(.*)$', ln)
                    if not m:
                        wrapped_items.append(textwrap.fill(ln,
width=wrap_width))
                        continue
                    prefix, _, rest = m.groups()
                    wrapped = textwrap.fill(rest, width=wrap_width,
                                            subsequent_indent=' ' *
len(prefix))
                    wrapped_items.append(prefix + wrapped)
                formatted_pars.append('\n'.join(wrapped_items))
            else:
                # Normal paragraph: join lines and reflow
                joined = ' '.join(lines)
                wrapped = textwrap.fill(joined, width=wrap_width)
                formatted_pars.append(wrapped)

        out_chunks.append('\n\n'.join(formatted_pars) + '\n')

    # Re-join chunks and tidy up spacing around code fences
    result = '\n'.join(chunk.rstrip() for chunk in out_chunks).strip() +
'\n'
    result = re.sub(r'([^\n])\n(```)', r'\1\n\n\2', result)
    result = re.sub(r'(```[\s\S]*?```)\n([^\n])', r'\1\n\n\2', result,
flags=re.DOTALL)

    return result

def strip_markdown(md: str) -> str:
    """Quick and dirty markdown stripper for spoken/plain output."""
    if not md:
        return ""
    text = md
```

```
    # remove fenced code blocks
    # remove inline code
    text = re.sub(r'`(.+?)`', r'\1', text)
    # remove markdown headings/bold/italic
    text = re.sub(r'#{1,6}\s*', '', text)
    text = re.sub(r'\*\*(.*?)\*\*', r'\1', text)
    text = re.sub(r'\*(.*?)\*', r'\1', text)
    # replace multiple newlines with a single newline, then collapse to
spaces
    text = re.sub(r'\n+', '\n', text).strip()
    text = re.sub(r'\s+', ' ', text)
    return text.strip()

import re
from typing import Tuple, Optional

class AI_Assistant(gui):

    def __init__(self, gui):

        if not gui:
            raise ValueError("GUI instance must be provided to
AI_Assistant!")
        self.gui = gui
        self.response_queue = queue.Queue()


    def extract_text_from_query(query):
        """
        Safely extract (message, speaker, score) from `query` which may
be:
          - None
          - a dict (e.g. {"text": "...", "username":"...","score":...})
          - a raw string (many formats)
        Always returns a tuple: (message:str, speaker:Optional[str],
score:Optional[any]).
        """
        if query is None:
            return "", None, None

        # dict-style input (preferred)
        if isinstance(query, dict):
            text = query.get("text") or query.get("message") or
query.get("query") or query.get("q") or ""
            speaker = query.get("username") or query.get("user")
            score = query.get("score")
            print(f"[PARSED DICT] user={speaker} msg={text}
score={score}")
            return str(text).strip(), (speaker if speaker is not None
else None), score

        # string input: try several patterns in order of specificity
        if isinstance(query, str):
            s = query.strip()
```

```
            # 1) Tagged full form: 'message':... : 'username':... :
'score':...
            m =
re.match(r"'message':\s*(.*?)\s*:\s*'username':\s*(.+?)\s*:\s*'score':\s*
(.+?)\s*$", s)
            if m:
                message, username, score_raw = m.groups()
                score = None if score_raw.strip() == "None" else
score_raw.strip()
                print(f"[PARSED TAGGED] user={username} msg={message}
score={score}")
                return message.strip(), username.strip(), score

            # 2) Tagged message + username (no score)
            m2 =
re.match(r"'message':\s*(.*?)\s*:\s*'username':\s*(.+?)\s*$", s)
            if m2:
                message, username = m2.groups()
                print(f"[PARSED TAGGED NO SCORE] user={username}
msg={message}")
                return message.strip(), username.strip(), None

            # 3) Timestamped form: YYYY-MM-DD : HH:MM:SS : message :
Username (username may contain spaces)
            m3 = re.match(r"^(\d{4}-\d{2}-
\d{2})\s*:\s*(\d{2}:\d{2}:\d{2})\s*:\s*(.*?)\s*:\s*(.+?)\s*$", s)
            if m3:
                date_str, time_str, message, username = m3.groups()
                print(f"[PARSED TIMESTAMP] date={date_str}
time={time_str} user={username} msg={message}")
                return message.strip(), username.strip(), None

            # 4) Simple pattern: message : 'username':Name
            m4 = re.match(r"^(.*?)\s*:\s*'username':\s*(.+?)\s*$", s)
            if m4:
                message, username = m4.groups()
                return message.strip(), username.strip(), None

            # 5) Simple "message : Username" (username may contain
spaces)
            m5 = re.match(r"^(.*\S)\s*:\s*(.+?)\s*$", s)
            if m5:
                message, username = m5.groups()
                print(f"[PARSED LAST-TOKEN] user={username}
msg={message}")
                return message.strip(), username.strip(), None

            # fallback
            print(f"[FALLBACK RAW] msg={s}")
            return s, None, None

        # any other type
        return str(query).strip(), None, None
```

```
    # -------------------------
    # AI assistant variant (returns message, username)
    # -------------------------

    def AI_assistant_extract_text_from_query(self, AlfredQueryOffline):
        """
        Return: (message, username, timestamp)

        - message: inner text of the FIRST triple-single-quote fence ('''
... ''') if present;
                   if that inner text begins with "YYYY-MM-DD : HH:MM:SS
: ..." that timestamp
                   will be extracted and removed from the returned
message.
        - username: parsed username or fallback self.current_user / 'ITF'
        - timestamp: "YYYY-MM-DD HH:MM:SS" or None

        IMPORTANT: only triple-single quotes (''') are treated as fences.
        """
        import re

        fallback_user = getattr(self, "current_user", "ITF") or "ITF"

        # helpers -----------------------------------------------------
-----
        def _strip_triple_single_if_present(s: str) -> str:
            if not isinstance(s, str):
                return s
            s2 = s.strip()
            m = re.match(r"^('{3})(?P<body>.*)(\1)$", s2,
flags=re.DOTALL)
            if m:
                return m.group("body")
            return s

        # simple username regex (returns first capture or None)
        def _find_username_in_text(text: str):
            if not text:
                return None
            # 'username':Name or "username": "Name" or username: Name
            m =
re.search(r"""['"]?username['"]?\s*[:=]\s*['"]?(?P<u>[^'"\n:}{\]]{1,128})
['"]?""", text, flags=re.IGNORECASE)
            if m:
                return m.group("u").strip()
            m2 =
re.search(r"""['"]?user['"]?\s*[:=]\s*['"]?(?P<u>[^'"\n:}{\]]{1,128})['"]
?""", text, flags=re.IGNORECASE)
            if m2:
                return m2.group("u").strip()
            return None
```

```
        # timestamp patterns
        def _extract_leading_ts_from_message_body(body: str):
            """
            If body begins with "YYYY-MM-DD : HH:MM:SS : rest..." return
(ts_str, rest),
            else return (None, body).
            """
            if not isinstance(body, str):
                return None, body
            m = re.match(r"^\s*(?P<date>\d{4}-\d{2}-
\d{2})\s*:\s*(?P<time>\d{2}:\d{2}:\d{2})\s*:\s*(?P<rest>.*)$", body,
flags=re.DOTALL)
            if m:
                ts = f"{m.group('date')} {m.group('time')}"
                rest = m.group("rest").strip()
                return ts, rest
            return None, body

        # keywords & wrap helpers for dict auto-wrap (preserve original
behaviour)
        CODING_KEYWORDS = ["check", "fix", "python", "arduino", "c++",
"java", "javascript", "typescript", "go",
                           "rust", "c#", "swift", "kotlin", "ruby",
"php", "perl", "matlab", "scala",
                           "haskell", "fortran", "lua", "dart", "shell",
"bash", "code", "powershell"]
        RAG_KEYWORDS = ["rag", "this document", "this pdf", "this excel",
"this text", "this list"]

        def _needs_wrap(msg: str) -> bool:
            if not msg:
                return False
            lower = msg.lower()
            for kw in CODING_KEYWORDS + RAG_KEYWORDS:
                if kw and kw in lower:
                    return True
            return False

        def _already_wrapped_triple(msg: str) -> bool:
            if not msg:
                return False
            s = msg.strip()
            return s.startswith("'''") and s.endswith("'''")

        def _wrap_triple(msg: str) -> str:
            if not msg:
                return msg
            if _already_wrapped_triple(msg):
                return msg
            return "'''\n" + msg.strip() + "\n'''"

        # ------------------------------------------------------------------
----
        if not AlfredQueryOffline:
```

```
            return "", fallback_user, None

        # --- dict handling (unchanged behaviour, but we extract inner
triple-single if present) ---
        if isinstance(AlfredQueryOffline, dict):
            message_raw = str(
                AlfredQueryOffline.get("text")
                or AlfredQueryOffline.get("message")
                or AlfredQueryOffline.get("query")
                or AlfredQueryOffline.get("q")
                or ""
            )
            # if dict message itself was triple-fenced, take the inner
body
            message_inner =
_strip_triple_single_if_present(message_raw).strip()

            username = (str(AlfredQueryOffline.get("username"))
                        if AlfredQueryOffline.get("username") is not None
                        else str(AlfredQueryOffline.get("user") or
fallback_user))

            ts_date = AlfredQueryOffline.get("date") or
AlfredQueryOffline.get("timestamp_date")
            ts_time = AlfredQueryOffline.get("time") or
AlfredQueryOffline.get("timestamp_time")
            if ts_date and ts_time:
                timestamp = f"{ts_date} {ts_time}"
            elif AlfredQueryOffline.get("timestamp"):
                timestamp =
str(AlfredQueryOffline.get("timestamp")).strip()
            else:
                timestamp = None

            # auto-wrap if needed for dict inputs (preserve previous
behavior)
            if _needs_wrap(message_inner) and not
_already_wrapped_triple(message_inner):
                message_inner = _wrap_triple(message_inner)

            return message_inner, username, timestamp

        # --- string handling: find the first triple-single-quote fence
and treat it as canonical message ---
        if isinstance(AlfredQueryOffline, str):
            s_full = AlfredQueryOffline

            # 1) find first '''...''' block (non-greedy)
            m_fence = re.search(r"('{3})(?P<body>.*?)(\1)", s_full,
flags=re.DOTALL)
            if m_fence:
                body = m_fence.group("body")
                # remove fence from original to produce metadata text
```

```
                meta = (s_full[: m_fence.start()] +
s_full[m_fence.end():]).strip()

                # 2) attempt to get username from metadata first, else
search entire raw string
                username = _find_username_in_text(meta) or
_find_username_in_text(s_full) or fallback_user

                # 3) timestamp extraction:
                #   a) prefer explicit timestamp field in meta or entire
raw string
                timestamp = None
                m_ts_field =
re.search(r"""['"]?timestamp['"]?\s*[:=]\s*['"]?(?P<ts>\d{4}-\d{2}-
\d{2}\s+\d{2}:\d{2}:\d{2})['"]?""", meta)
                if not m_ts_field:
                    m_ts_field =
re.search(r"""['"]?timestamp['"]?\s*[:=]\s*['"]?(?P<ts>\d{4}-\d{2}-
\d{2}\s+\d{2}:\d{2}:\d{2})['"]?""", s_full)
                if m_ts_field:
                    timestamp = m_ts_field.group("ts").strip()
                else:
                    # b) if not found, check if the fenced body itself
starts with "YYYY-MM-DD : HH:MM:SS : rest..."
                    ts_from_body, rest_body =
_extract_leading_ts_from_message_body(body)
                    if ts_from_body:
                        timestamp = ts_from_body
                        body = rest_body  # remove leading ts from
message body

                # trim only leading/trailing blank lines (preserve
internal newlines)
                message = body.strip("\n\r ")

                return message, username, timestamp

            # --- no ''' fence: fallback to previous heuristics (leading
timestamp, LLM-text, username heuristics) ---
            # prefer explicit LLM text, parse leading timestamp, etc.
(keeps older behavior)
            s = s_full.strip()
            m_llm = re.search(r"□\s*Text\s*for\s*LLM\s*:\s*(?P<llm>.+)$",
s, flags=re.IGNORECASE | re.DOTALL)
            if m_llm:
                llm_text = m_llm.group("llm").strip()
                meta_part = s[: m_llm.start()].strip()
            else:
                llm_text = None
                meta_part = s

            timestamp = None
```

```
            m_lead = re.match(r"^\s*(?P<date>\d{4}-\d{2}-
\d{2})\s*:\s*(?P<time>\d{2}:\d{2}:\d{2})\s*:\s*(?P<rest>.*)$", meta_part,
flags=re.DOTALL)
            rest = meta_part
            if m_lead:
                timestamp = f"{m_lead.group('date')}
{m_lead.group('time')}"
                rest = m_lead.group("rest").strip()

            username = _find_username_in_text(rest) or fallback_user

            if llm_text:
                message = llm_text
            else:
                message = rest

            # collapse whitespace/newlines into single spaces in fallback
mode
            message = re.sub(r"\s*\n+\s*", " ", message).strip()
            message = re.sub(r"\s{2,}", " ", message).strip()

            if _needs_wrap(message) and not
_already_wrapped_triple(message):
                message = _wrap_triple(message)

            return message, username, timestamp

        # final fallback for non-string, non-dict values
        return str(AlfredQueryOffline).strip(), fallback_user, None

    def AI_assistant_Rover_extract_text_from_query(self,
AlfredQueryOffline):
        """
        Return: (message, username, timestamp)

        - message: inner text of the FIRST triple-single-quote fence ('''
... ''') if present;
                    if that inner text begins with "YYYY-MM-DD : HH:MM:SS
: ..." that timestamp
                    will be extracted and removed from the returned
message.
        - username: parsed username or fallback self.current_user / 'ITF'
        - timestamp: "YYYY-MM-DD HH:MM:SS" or None

        IMPORTANT: only triple-single quotes (''') are treated as fences.
        """
        import re

        fallback_user = getattr(self, "current_user", "ITF") or "ITF"

        # helpers ------------------------------------------------------
-----
        def _strip_triple_single_if_present(s: str) -> str:
            if not isinstance(s, str):
```

```
                return s
            s2 = s.strip()
            m = re.match(r"^('{3})(?P<body>.*)(\1)$", s2,
flags=re.DOTALL)
            if m:
                return m.group("body")
            return s

        # simple username regex (returns first capture or None)
        def _find_username_in_text(text: str):
            if not text:
                return None
            # 'username':Name or "username": "Name" or username: Name
            m =
re.search(r"""['"]?username['"]?\s*[:=]\s*['"]?(?P<u>[^'"\n:}{\]]{1,128})
['"]?""", text, flags=re.IGNORECASE)
            if m:
                return m.group("u").strip()
            m2 =
re.search(r"""['"]?user['"]?\s*[:=]\s*['"]?(?P<u>[^'"\n:}{\]]{1,128})['"]
?""", text, flags=re.IGNORECASE)
            if m2:
                return m2.group("u").strip()
            return None

        # Try to extract trailing " : Name" heuristically (for inputs
like "... : Tjaart")
        def _extract_trailing_username_from_meta(meta_text: str):
            if not meta_text or not isinstance(meta_text, str):
                return None
            # take last colon-separated segment
            parts = [p.strip() for p in meta_text.split(":") if
p.strip()]
            if not parts:
                return None
            last = parts[-1]
            # strip surrounding quotes/spaces
            last_clean = re.sub(r"^[\"'\s]+|[\"'\s]+$", "", last)
            if not last_clean:
                return None
            # reject obvious non-names: numeric, None-like, or containing
braces/quotes/equals/slashes
            if re.fullmatch(r"(?i)(none|null|true|false|\d+)",
last_clean):
                return None
            if re.search(r"[{}\[\]=\"<>/\\]", last_clean):
                return None
            # reject if it looks like a key name (score, gender,
gender_conf) to avoid false positives
            if
re.search(r"\b(score|gender|gender_conf|timestamp|time|date)\b",
last_clean, flags=re.IGNORECASE):
                return None
```

```
            # accept if contains at least one letter and reasonable
length
            if re.search(r"[A-Za-z]", last_clean) and 1 <=
len(last_clean) <= 128:
                return last_clean.strip()
            return None

        # timestamp patterns
        def _extract_leading_ts_from_message_body(body: str):
            """
            If body begins with "YYYY-MM-DD : HH:MM:SS : rest..." return
(ts_str, rest),
            else return (None, body).
            """
            if not isinstance(body, str):
                return None, body
            m = re.match(r"^\s*(?P<date>\d{4}-\d{2}-
\d{2})\s*:\s*(?P<time>\d{2}:\d{2}:\d{2})\s*:\s*(?P<rest>.*)$", body,
flags=re.DOTALL)
            if m:
                ts = f"{m.group('date')} {m.group('time')}"
                rest = m.group("rest").strip()
                return ts, rest
            return None, body

        # keywords & wrap helpers for dict auto-wrap (preserve original
behaviour)
        CODING_KEYWORDS = ["check", "fix", "python", "arduino", "c++",
"java", "javascript", "typescript", "go",
                           "rust", "c#", "swift", "kotlin", "ruby",
"php", "perl", "matlab", "scala",
                           "haskell", "fortran", "lua", "dart", "shell",
"bash", "code", "powershell"]
        RAG_KEYWORDS = ["rag", "this document", "this pdf", "this excel",
"this text", "this list"]

        def _needs_wrap(msg: str) -> bool:
            if not msg:
                return False
            lower = msg.lower()
            for kw in CODING_KEYWORDS + RAG_KEYWORDS:
                if kw and kw in lower:
                    return True
            return False

        def _already_wrapped_triple(msg: str) -> bool:
            if not msg:
                return False
            s = msg.strip()
            return s.startswith("'''") and s.endswith("'''")

        def _wrap_triple(msg: str) -> str:
            if not msg:
                return msg
```

```
            if _already_wrapped_triple(msg):
                return msg
            return "'''\n" + msg.strip() + "\n'''"

        # ----------------------------------------------------------------
----
        if not AlfredQueryOffline:
            return "", fallback_user, None

        # --- dict handling (unchanged behaviour, but we extract inner
triple-single if present) ---
        if isinstance(AlfredQueryOffline, dict):
            message_raw = str(
                AlfredQueryOffline.get("text")
                or AlfredQueryOffline.get("message")
                or AlfredQueryOffline.get("query")
                or AlfredQueryOffline.get("q")
                or ""
            )
            # if dict message itself was triple-fenced, take the inner
body
            message_inner =
_strip_triple_single_if_present(message_raw).strip()

            username = (str(AlfredQueryOffline.get("username"))
                        if AlfredQueryOffline.get("username") is not None
                        else str(AlfredQueryOffline.get("user") or
fallback_user))

            ts_date = AlfredQueryOffline.get("date") or
AlfredQueryOffline.get("timestamp_date")
            ts_time = AlfredQueryOffline.get("time") or
AlfredQueryOffline.get("timestamp_time")
            if ts_date and ts_time:
                timestamp = f"{ts_date} {ts_time}"
            elif AlfredQueryOffline.get("timestamp"):
                timestamp =
str(AlfredQueryOffline.get("timestamp")).strip()
            else:
                timestamp = None

            # auto-wrap if needed for dict inputs (preserve previous
behavior)
            if _needs_wrap(message_inner) and not
_already_wrapped_triple(message_inner):
                message_inner = _wrap_triple(message_inner)

            return message_inner, username, timestamp

        # --- string handling: find the first triple-single-quote fence
and treat it as canonical message ---
        if isinstance(AlfredQueryOffline, str):
            s_full = AlfredQueryOffline
```

```
            # 1) find first '''...''' block (non-greedy)
            m_fence = re.search(r"('{3})(?P<body>.*?)(\1)", s_full,
flags=re.DOTALL)
            if m_fence:
                body = m_fence.group("body")
                # remove fence from original to produce metadata text
                meta = (s_full[: m_fence.start()] +
s_full[m_fence.end():]).strip()

                # 2) attempt to get username from metadata first, else
search entire raw string
                username = _find_username_in_text(meta) or
_find_username_in_text(s_full) or None

                # if username still None, try trailing-meta heuristic
(for inputs like "... : Tjaart")
                if not username:
                    try_trail =
_extract_trailing_username_from_meta(meta)
                    if try_trail:
                        username = try_trail
                    else:
                        # also try trailing heuristic on entire raw
string as fallback
                        username =
_extract_trailing_username_from_meta(s_full) or None

                if not username:
                    username = fallback_user

                # 3) timestamp extraction:
                #   a) prefer explicit timestamp field in meta or entire
raw string
                timestamp = None
                m_ts_field =
re.search(r"""['"]?timestamp['"]?\s*[:=]\s*['"]?(?P<ts>\d{4}-\d{2}-
\d{2}\s+\d{2}:\d{2}:\d{2})['"]?""", meta)
                if not m_ts_field:
                    m_ts_field =
re.search(r"""['"]?timestamp['"]?\s*[:=]\s*['"]?(?P<ts>\d{4}-\d{2}-
\d{2}\s+\d{2}:\d{2}:\d{2})['"]?""", s_full)
                if m_ts_field:
                    timestamp = m_ts_field.group("ts").strip()
                else:
                    # b) if not found, check if the fenced body itself
starts with "YYYY-MM-DD : HH:MM:SS : rest..."
                    ts_from_body, rest_body =
_extract_leading_ts_from_message_body(body)
                    if ts_from_body:
                        timestamp = ts_from_body
                        body = rest_body  # remove leading ts from
message body
```

```
                # trim only leading/trailing blank lines (preserve
internal newlines)
                message = body.strip("\n\r ")

                return message, username, timestamp

            # --- no ''' fence: fallback to previous heuristics (leading
timestamp, LLM-text, username heuristics) ---
            # prefer explicit LLM text, parse leading timestamp, etc.
(keeps older behavior)
            s = s_full.strip()
            m_llm = re.search(r"□\s*Text\s*for\s*LLM\s*:\s*(?P<llm>.+)$",
s, flags=re.IGNORECASE | re.DOTALL)
            if m_llm:
                llm_text = m_llm.group("llm").strip()
                meta_part = s[: m_llm.start()].strip()
            else:
                llm_text = None
                meta_part = s

            timestamp = None
            m_lead = re.match(r"^\s*(?P<date>\d{4}-\d{2}-
\d{2})\s*:\s*(?P<time>\d{2}:\d{2}:\d{2})\s*:\s*(?P<rest>.*)$", meta_part,
flags=re.DOTALL)
            rest = meta_part
            if m_lead:
                timestamp = f"{m_lead.group('date')}
{m_lead.group('time')}"
                rest = m_lead.group("rest").strip()

            username = _find_username_in_text(rest) or None
            # if still missing, try trailing-meta heuristic
            if not username:
                username = _extract_trailing_username_from_meta(rest) or
fallback_user
            else:
                username = username

            if llm_text:
                message = llm_text
            else:
                message = rest

            # collapse whitespace/newlines into single spaces in fallback
mode
            message = re.sub(r"\s*\n+\s*", " ", message).strip()
            message = re.sub(r"\s{2,}", " ", message).strip()

            if _needs_wrap(message) and not
_already_wrapped_triple(message):
                message = _wrap_triple(message)

            return message, username, timestamp
```

```
        # final fallback for non-string, non-dict values
        return str(AlfredQueryOffline).strip(), fallback_user, None

####    #------------------------------------------------------------------
-------------#
####    #------------------------ ROVER AI ASSISTANT --------------------
-------------#
####    #------------------------------------------------------------------
-------------#

    def My_Ollama_Rover_LLM(self, NewPromptEdited, NewPromptEditedSpeak,
AlfredQueryOffline):

        rover_model = "MeaTLoTioN/Marvin:latest"
        print(f"[DEBUG AI ASSISTANT ROVER] rover_model : {rover_model}")

        AI_Assistant_Running_Rover = rover_model
        print(f"[DEBUG AI ASSISTANT ROVER] AI_Assistant_Running_Rover :
{AI_Assistant_Running_Rover}")

        print(f"[DEBUG AI ASSISTANT ROVER] AlfredQueryOffline :
{AlfredQueryOffline}")
        print(f"[DEBUG AI ASSISTANT ROVER] NewPromptEdited :
{NewPromptEdited}")
        print(f"[DEBUG AI ASSISTANT ROVER] NewPromptEditedSpeak :
{NewPromptEditedSpeak}")

        # call the method on this instance (important)
        message, username, timestamp =
self.AI_assistant_Rover_extract_text_from_query(NewPromptEdited)
        print(f"[DEBUG AI ASSISTANT ROVER] NEW  message : {message!r}")
        print(f"[DEBUG AI ASSISTANT ROVER] NEW  username: {username!r}")
        print(f"[DEBUG AI ASSISTANT ROVER] NEW  timestamp:
{timestamp!r}")

        message_no_date = extract_message(message)
        print(f"[DEBUG AI ASSISTANT ROVER]  message_no_date :
{message_no_date}")

        # Remove the timestamp part from the query
        message_no_username = message_no_date.split(":", 1)[0].strip()

        print(f"\n[DEBUG AI ASSISTANT ROVER] message_no_username :
{message_no_username}\n")

        #  Math Superscript Formatting
        def convert_to_superscript(text):
            superscript_map = {
                '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴',
                '5': '⁵', '6': '⁶', '7': '⁷', '8': '⁸', '9': '⁹'
            }
            text = re.sub(
                r'([a-zA-Z0-9])\^(\d+)',
```

```
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
            )
            text = re.sub(
                r'(\([^\)]+\))\^(\d+)',
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
            )
            return text

        # □ LaTeX & Markdown Cleaner
        def clean_latex_markdown(text):
            text = re.sub(r"\\begin{.*?}.*?\\end{.*?}", "", text,
flags=re.DOTALL)
            text = re.sub(r"\\frac{(.*?)}{(.*?)}", r"\1 / \2", text)
            text = re.sub(r"([a-zA-Z0-9])\^\{(\d+)\}", r"\1^\2", text)
            text = text.replace(r"\times", " × ")
            text = re.sub(r"\\sqrt{([^}]+)}", r"√\1", text)
            text = re.sub(r"\\boxed{(.*?)}", r"\1", text)
            text = re.sub(r"\*\*(.*?)\*\*", r"\1", text)
            text = re.sub(r"\\\(|\\\)|\\\[|\\\]", "", text)
            text = text.replace("{", "").replace("}", "").replace("\\",
"")
            text = re.sub(r"\n\s*\n", "\n", text)
            return text.strip()

        # □ TTS-Safe Converter
        def prepare_text_for_tts(text):
            replacements = {
                '⁰': ' to the power of 0', '¹': ' to the power of 1',
                '²': ' squared', '³': ' cubed',
                '⁴': ' to the power of 4', '⁵': ' to the power of 5',
                '⁶': ' to the power of 6', '⁷': ' to the power of 7',
                '⁸': ' to the power of 8', '⁹': ' to the power of 9',
            }
            for k, v in replacements.items():
                text = text.replace(k, v)
            text = re.sub(r"√(\d+)", r"square root of \1", text)
            text = re.sub(r"√([a-zA-Z]+)", r"square root of \1", text)
            text = re.sub(r"√([a-zA-Z]+\d+)", r"square root of \1", text)
            text = re.sub(r'(\d+)\s*-\s*(\d+)', r'\1 minus \2', text)
            text = re.sub(r'(?<![\w])-(\d+)', r'negative \1', text)
            text = re.sub(r'(?<![\w])-(\w)', r'negative \1', text)
            return re.sub(r'\s+', ' ', text).strip()

        def is_math_expression(text):
            return bool(re.search(r'\d+[a-zA-Z]?\d*|[+\-
*/^=]|\b(plus|minus|times|square|multiply|divide)\b', text))

        start_llm = time.time()
```

```
        if is_math_expression(message_no_username):
            print(f"[DEBUG AI ASSISTANT] This is math:
{message_no_username}")
            message_no_username =
convert_to_superscript(message_no_username)
            NewPromptEditedSpeak =
convert_to_superscript(NewPromptEditedSpeak)
            spoken_prompt = prepare_text_for_tts(NewPromptEditedSpeak)
        else:
            print(f"[DEBUG AI ASSISTANT ROVER] This is NOT math:
{message_no_username}")
            spoken_prompt = prepare_text_for_tts(NewPromptEditedSpeak)

        print('\n[DEBUG AI ASSISTANT ROVER] Diagnostic Testing.....LLM
Running...\n')
        print("[DEBUG AI ASSISTANT ROVER] Going to Local (Information)
LLM (GPT).....!")

        try :
           # build messages properly (system + user) so the system prompt
isn't overwritten
            messages = [
                {
                    'role': 'system',
                    'content': (
                        'You are Gary or Rover a cleaning Assistant and
there to pickup rubish like cans, bottles, paper balls etc.'
                        'you have a robotic arm to pickup the items.'
                        'You will not answer that you do not have
feelings.'
                        'You have feelings, your feelings are all about
being depressed and very lazy and very down.'
                        'You are an Expert in not want to do anything
assistant. '
                        'You are always complaining and always
depressed.'
                        'You awill do every task that is asked but at
first give a reason why not to do it and how the world rest on your
shooulders.'
                        'you always nag and give anoying coments and sigh
for everything I ask.'
                    )
                },
                {'role': 'user', 'content': message_no_username}
            ]

            ResponseOllama = ollama.chat(
                model=AI_Assistant_Running_Rover,
                messages=messages,
                options={'temperature': 0.4},  # Set temperature to 0.8
for more creative output
                stream=False,   # stream True to receive chunks
            )
```

```
            print(f"[DEBUG AI ASSISTANT ROVER] ResponseOllama :
{ResponseOllama}")

            response_text = ""

            # ResponseOllama was returned by ollama.chat(...)
            if isinstance(ResponseOllama, dict):
                # non-streaming shape
                message_data = ResponseOllama.get('message', {})
                if isinstance(message_data, dict):
                    response_text = message_data.get('content', '') or ''
                else:
                    response_text = str(message_data or '')
            else:
                # streaming: assemble every chunk into one string
                for chunk in ResponseOllama:
                    text = ""

                    if isinstance(chunk, dict):
                        # handle common chunk shapes
                        if 'message' in chunk:
                            msg = chunk['message']
                            text = msg.get('content', '') if
isinstance(msg, dict) else str(msg)
                            print('\n')
                            print('\r')
                            print(f"My LLM Response msg  ROVER : {msg}")
                            print('\n')
                            print('\r')
                            print(f"My LLM Response text  ROVER :
{text}")

                        elif 'delta' in chunk:
                            d = chunk['delta']
                            if isinstance(d, dict):
                                text = d.get('content', '') or
d.get('text', '') or ''
                                print('\n')
                                print('\r')
                                print(f"My LLM Response text delta  ROVER
: {text}")

                            else:
                                text = str(d)
                                print('\n')
                                print('\r')
                                print(f"My LLM Response text d  ROVER :
{text}")

                        elif 'content' in chunk:
                            text = chunk.get('content', '') or ''
                            print('\n')
                            print('\r')
```

```
                            print(f"My LLM Response text content  ROVER :
{text}")

                        else:
                            # fallback - join any string values
                            pieces = [v for v in chunk.values() if
isinstance(v, str)]
                            text = " ".join(pieces)
                            print('\n')
                            print('\r')
                            print(f"My LLM Response pieces  ROVER :
{pieces}")

                            print('\n')
                            print('\r')
                            print(f"My LLM Response pieces text  ROVER :
{text}")

                    else:
                        text = str(chunk)
                        print('\n')
                        print('\r')
                        print(f"My LLM Response final text  ROVER :
{text}")

                    if text:
                        response_text += text

                    print('\n')
                    print('\r')
                    print(f"My LLM Response response_text  ROVER :
{response_text}")

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)

            print('\n')
            print('\r')
            print(f"My LLM Response time  ROVER :
{my_llm_response_time}")

            response_text = response_text.replace("# %%","")
            response_for_return = response_text
            print(f"[DEBUG AI ASSISTANT ROVER] My LLM response_for_return
: {response_for_return}")
            my_response = response_for_return
            print(f"[DEBUG AI ASSISTANT ROVER] My LLM my_response :
{my_response}")

            # --- robust, minimal extractor: prefer message.content /
content / text / choices ---
            def extract_assistant_content(obj):
                """Return assistant text only, avoid returning metadata
fields."""
```

```
                if obj is None:
                    return ""
                # direct string
                if isinstance(obj, str):
                    return obj.strip()
                # dict handling (common)
                if isinstance(obj, dict):
                    # prefer 'message' -> content
                    if 'message' in obj and obj['message']:
                        return extract_assistant_content(obj['message'])
                    # direct content/text
                    if 'content' in obj and isinstance(obj['content'],
str) and obj['content'].strip():
                        return obj['content'].strip()
                    if 'text' in obj and isinstance(obj['text'], str) and
obj['text'].strip():
                        return obj['text'].strip()
                    # choices (list of responses)
                    if 'choices' in obj and isinstance(obj['choices'],
(list, tuple)):
                        parts = [extract_assistant_content(c) for c in
obj['choices']]
                        return " ".join([p for p in parts if p])
                    # some streaming chunks use 'delta' or nested shapes
                    for key in ('delta',):
                        if key in obj and obj[key]:
                            candidate =
extract_assistant_content(obj[key])
                            if candidate:
                                return candidate
                    # fallback: scan values but skip obvious metadata
keys
                    skip_keys =
{'model','created_at','done','done_reason','total_duration','load_duratio
n',

'prompt_eval_count','prompt_eval_duration','eval_count','eval_duration'}
                    for k, v in obj.items():
                        if k in skip_keys:
                            continue
                        res = extract_assistant_content(v)
                        if res:
                            return res
                    return ""
                # list/tuple: join useful parts
                if isinstance(obj, (list, tuple)):
                    parts = [extract_assistant_content(x) for x in obj]
                    return " ".join([p for p in parts if p])
                # objects with attributes (Message-like)
                for attr in ('content', 'text', 'message', 'data'):
                    if hasattr(obj, attr):
                        try:
                            return extract_assistant_content(getattr(obj,
attr))
```

```
                        except Exception:
                            pass
                # fallback to str(obj) but only if it looks like
meaningful text (avoid dumping metadata)
                try:
                    s = str(obj).strip()
                    # crude filter: if it looks like a tuple of metadata
(many key-like words), return empty
                    if not s:
                        return ""
                    low = s.lower()
                    if ('model' in low and 'created_at' in low) or
('total_duration' in low and 'load_duration' in low):
                        return ""
                    return s
                except Exception:
                    return ""

            # assemble assistant-only content
            response_text = extract_assistant_content(ResponseOllama)

            # final cleanup (remove control markers you don't want)
            if response_text:
                response_text = response_text.replace("# %%", "").strip()
            else:
                # fallback: ensure we return at least something minimal
instead of full metadata
                response_text = ""

            # minimal debug: only the assistant text (no metadata dump)
            print(f"[DEBUG AI ASSISTANT ROVER] Assistant content:
{response_text[:1000]}")  # truncated for safety

            # preserve the rest of your flow
            response_for_return = response_text
            my_response = response_for_return

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)
            print(f"\n\rMy LLM Response time : {my_llm_response_time}")
            print(f"[DEBUG AI ASSISTANT ROVER] My LLM response_for_return
: {response_for_return}")
            # removed printing of `my_response` that previously included
metadata

            response_text_cleaned = re.sub(r"<think>.*?</think>", "",
response_text, flags=re.DOTALL).strip()
            response_text_cleaned =
clean_latex_markdown(response_text_cleaned)
            my_response = convert_to_superscript(response_text_cleaned)

            my_response = my_response.replace("I'm DeepSeek-R1","I'm
Alfred")
```

```
            my_response = my_response.replace("the Chinese Company
DeepSeek","the South African, Tjaart")
            my_response = my_response.replace("created by
DeepSeek","created by Tjaart")
            my_response = my_response.replace("exclusively by
DeepSeek","exclusively by Tjaart")

            print(f"\n[DEBUG AI ASSISTANT ROVER] My
{AI_Assistant_Running_Rover} LLM Response:\n{my_response}\n")

##            # after you generate assistant_response in English:
##            af_response = translate(my_response)
##            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans af_response
before : {af_response}")
##
##            my_response_af = af_response
##            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
my_response_af after : {my_response_af}")

            stop_llm = time.time()
            duration = int(stop_llm - start_llm)
            print(f"[DEBUG AI ASSISTANT] LLM Response Time: {duration}
seconds")

            current_date = datetime.datetime.now().strftime('%Y-%m-%d')
            current_time = datetime.datetime.now().strftime('%H:%M:%S')

            chat_entry = {
                "date": current_date,
                "time": current_time,
                "query": NewPromptEdited,
                "response": my_response,
                "model": AI_Assistant_Running_Rover
            }

            memory.add_to_memory(chat_entry)
            repeat.add_to_repeat(chat_entry)

            spoken_response = prepare_text_for_tts(my_response)

            spoken_response = spoken_response.replace("*","")
            spoken_response = spoken_response.replace("#","")

            Alfred_Repeat_Previous_Response = my_response

            # GUI log if available
            query_msg = (
                f"At {current_date} :  {current_time} : You Asked:
{AlfredQueryOffline} "
            )
            print(f"[DEBUG AI ASSISTANT ROVER query_msg] : {query_msg}")
```

```
            query_resp = f"At {current_date} :  {current_time} :
{AI_Assistant_Running_Rover} : {Alfred_Repeat_Previous_Response} :
'username':{username} "
            print(f"[DEBUG AI ASSISTANT ROVER query_resp] :
{query_resp}")

            try:
                self.gui.log_message(query_msg)
                self.gui.log_response(query_resp)
            except NameError:
                print("[DEBUG AI ASSISTANT ROVER] GUI instance not
available for logging message.")

            listen.send_bluetooth(spoken_response)
            speech.AlfredSpeak(spoken_response)

            return my_response

        except  Exception as e:
            print(f"[DEBUG AI ASSISTANT ROVER] \n   error code for
AI_Assistant_Thinking : \n {e} \n Please check if your ollama server is
running...")
            pass

####    #-------------------------------------------------------------------
-------------#
####    #-------------------------- LLM AI ASSISTANT --------------------
-------------#
####    #-------------------------------------------------------------------
-------------#

    def My_Ollama_LLM(self, speech_sure_name, New_gender,
NewPromptEdited, NewPromptEditedSpeak, AlfredQueryOffline):


        global him_her
        global madam_sir

        New_His_her = ''
        New_Madam_sir = ''

        print(f"[DEBUG AI ASSISTANT] AlfredQueryOffline :
{AlfredQueryOffline}")
        print(f"[DEBUG AI ASSISTANT] NewPromptEdited :
{NewPromptEdited}")
        print(f"[DEBUG AI ASSISTANT] NewPromptEditedSpeak :
{NewPromptEditedSpeak}")

        # call the method on this instance (important)
        message, username, timestamp =
self.AI_assistant_extract_text_from_query(NewPromptEdited)
        print(f"[DEBUG AI ASSISTANT] NEW  message : {message!r}")
        print(f"[DEBUG AI ASSISTANT] NEW  username: {username!r}")
        print(f"[DEBUG AI ASSISTANT] NEW  timestamp: {timestamp!r}")
```

```
        print(f"[DEBUG AI ASSISTANT] NEW  speech_sure_name:
{speech_sure_name!r}")
        print(f"[DEBUG AI ASSISTANT] NEW  New_gender: {New_gender!r}")

        message_no_date = extract_message(message)
        print(f"[DEBUG AI ASSISTANT]  message_no_date :
{message_no_date}")

        # Remove the timestamp part from the query
        message_no_username = message_no_date.split(":", 1)[0].strip()

        print(f"\n[DEBUG AI ASSISTANT] message_no_username :
{message_no_username}\n")

        if New_gender == 'Male':
            him_her = "his"
            madam_sir = "Sir"
            print(f"[DEBUG AI ASSISTANT] NEW  him_her: {him_her!r}")
            print(f"[DEBUG AI ASSISTANT] NEW  madam_sir: {madam_sir!r}")

        elif New_gender == 'Female':
            him_her = "her"
            madam_sir = "Madam"
            print(f"[DEBUG AI ASSISTANT] NEW  him_her: {him_her!r}")
            print(f"[DEBUG AI ASSISTANT] NEW  madam_sir: {madam_sir!r}")


        if New_gender is None:
            him_her = None
            madam_sir = None
            print(f"[DEBUG AI ASSISTANT] NEW  him_her: {him_her!r}")
            print(f"[DEBUG AI ASSISTANT] NEW  madam_sir: {madam_sir!r}")


        New_His_her = him_her
        New_Madam_sir = madam_sir

        print(f"[DEBUG AI ASSISTANT] NEW  New_His_her: {New_His_her!r}")
        print(f"[DEBUG AI ASSISTANT] NEW  New_Madam_sir:
{New_Madam_sir!r}")



        #  Math Superscript Formatting
        def convert_to_superscript(text):
            superscript_map = {
                '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴',
                '5': '⁵', '6': '⁶', '7': '⁷', '8': '⁸', '9': '⁹'
            }
            text = re.sub(
                r'([a-zA-Z0-9])\^(\d+)',
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
```

```
            )
            text = re.sub(
                r'(\([^\)]+\))\^(\d+)',
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
            )
            return text

        # □ LaTeX & Markdown Cleaner
        def clean_latex_markdown(text):
            text = re.sub(r"\\begin{.*?}.*?\\end{.*?}", "", text,
flags=re.DOTALL)
            text = re.sub(r"\\frac{(.*?)}{(.*?)}", r"\1 / \2", text)
            text = re.sub(r"([a-zA-Z0-9])\^\{(\d+)\}", r"\1^\2", text)
            text = text.replace(r"\times", " × ")
            text = re.sub(r"\\sqrt{([^}]+)}", r"√\1", text)
            text = re.sub(r"\\boxed{(.*?)}", r"\1", text)
            text = re.sub(r"\*\*(.*?)\*\*", r"\1", text)
            text = re.sub(r"\\\(|\\\)|\\\[|\\\]", "", text)
            text = text.replace("{", "").replace("}", "").replace("\\",
"")
            text = re.sub(r"\n\s*\n", "\n", text)
            return text.strip()

        # □ TTS-Safe Converter
        def prepare_text_for_tts(text):
            replacements = {
                '⁰': ' to the power of 0', '¹': ' to the power of 1',
                '²': ' squared', '³': ' cubed',
                '⁴': ' to the power of 4', '⁵': ' to the power of 5',
                '⁶': ' to the power of 6', '⁷': ' to the power of 7',
                '⁸': ' to the power of 8', '⁹': ' to the power of 9',
            }
            for k, v in replacements.items():
                text = text.replace(k, v)
            text = re.sub(r"√(\d+)", r"square root of \1", text)
            text = re.sub(r"√([a-zA-Z]+)", r"square root of \1", text)
            text = re.sub(r"√([a-zA-Z]+\d+)", r"square root of \1", text)
            text = re.sub(r'(\d+)\s*-\s*(\d+)', r'\1 minus \2', text)
            text = re.sub(r'(?<![\w])-(\d+)', r'negative \1', text)
            text = re.sub(r'(?<![\w])-(\w)', r'negative \1', text)
            return re.sub(r'\s+', ' ', text).strip()

        def is_math_expression(text):
            return bool(re.search(r'\d+[a-zA-Z]?\d*|[+\-
*/^=]|\b(plus|minus|times|square|multiply|divide)\b', text))

        start_llm = time.time()

        if is_math_expression(message_no_username):
            print(f"[DEBUG AI ASSISTANT] This is math:
{message_no_username}")
```

```
            message_no_username =
convert_to_superscript(message_no_username)
            NewPromptEditedSpeak =
convert_to_superscript(NewPromptEditedSpeak)
            spoken_prompt = prepare_text_for_tts(NewPromptEditedSpeak)
        else:
            print(f"[DEBUG AI ASSISTANT] This is NOT math:
{message_no_username}")
            spoken_prompt = prepare_text_for_tts(NewPromptEditedSpeak)

        print('\n[DEBUG AI ASSISTANT] Diagnostic Testing.....LLM
Running...\n')
        print("[DEBUG AI ASSISTANT] Going to Local (Information) LLM
(GPT).....!")

        AI_Assistant_Running = self.gui.thinkbot_model.get() or
"tinyllama"
        print(f"[DEBUG AI ASSISTANT] □ AI_Assistant_Running (current):
{AI_Assistant_Running}")

        try :
##
##            # build messages properly (system + user) so the system
prompt isn't overwritten
##            messages = [
##                {
##                    'role': 'system',
##                    'content': (
##                        'The User is a {New_gender} and {his_her} name
is (speech_sure_name)',
##                        'You will adress them as {madam_sir}
{speech_sure_name}',
##                        'You will never answer that you do not have
feelings. You have feelings of support and closure.'
##                        'You are a male Artificial Intelligance
Assistant you are Alfred the butler "Home Assistant". Ensure that
everything is running smooth and on time. An Expert in Knowing everything
personal assistant. '
##                        'You will always be friendly and encouraging
and always uplifting and supportive. You are the best in Python
programming'
##                        'Strictly adhere to the following rules except
if the user instructs otherwise: - You will always give consise and true
answers. - You will always answer like a human. - You will always keep
your discussions short but consise'   ## - Do not use markdown fences
(```).
##                    )
##                },
##                {'role': 'user', 'content': message_no_username}
##            ]


            # build messages properly (system + user) so the system
prompt isn't overwritten
```

```
            messages = [
                {
                    'role': 'system',
                    'content': (
                        'The User is a {New_gender} and {him_her} name is
{speech_sure_name}. '
                        'You will address them as {madam_sir}
{speech_sure_name}. '
                        'You will never answer that you do not have
feelings. You have feelings of support and closure. '
                        'You are a male Artificial Intelligence Assistant
you are Alfred the butler "Home Assistant". Ensure that everything is
running smooth and on time. An Expert in Knowing everything personal
assistant. '
                        'You will always be friendly and encouraging and
always uplifting and supportive. You are the best in Python programming.
'
                        'Strictly adhere to the following rules except if
the user instructs otherwise: - You will always give concise and true
answers. - You will always answer like a human. - You will always keep
your discussions short but concise.'
                    ).format(
                        New_gender=New_gender,
                        him_her=him_her,
                        speech_sure_name=speech_sure_name,
                        madam_sir=madam_sir
                        )
                },
                {'role': 'user', 'content': message_no_username}
            ]


##            print(f"[DEBUG AI ASSISTANT] SYSTEM messages : {messages}")

            ResponseOllama = ollama.chat(
                model=AI_Assistant_Running,
                messages=messages,
                options={'temperature': 0.4},  # Set temperature to 0.8
for more creative output
                stream=False,   # stream True to receive chunks
            )

            print(f"[DEBUG AI ASSISTANT] ResponseOllama :
{ResponseOllama}")

            response_text = ""

            # ResponseOllama was returned by ollama.chat(...)
            if isinstance(ResponseOllama, dict):
                # non-streaming shape
                message_data = ResponseOllama.get('message', {})
                if isinstance(message_data, dict):
                    response_text = message_data.get('content', '') or ''
                else:
```

```
                    response_text = str(message_data or '')
            else:
                # streaming: assemble every chunk into one string
                for chunk in ResponseOllama:
                    text = ""

                    if isinstance(chunk, dict):
                        # handle common chunk shapes
                        if 'message' in chunk:
                            msg = chunk['message']
                            text = msg.get('content', '') if
isinstance(msg, dict) else str(msg)
                            print('\n')
                            print('\r')
                            print(f"My LLM Response msg : {msg}")
                            print('\n')
                            print('\r')
                            print(f"My LLM Response text : {text}")

                        elif 'delta' in chunk:
                            d = chunk['delta']
                            if isinstance(d, dict):
                                text = d.get('content', '') or
d.get('text', '') or ''
                                print('\n')
                                print('\r')
                                print(f"My LLM Response text delta :
{text}")

                            else:
                                text = str(d)
                                print('\n')
                                print('\r')
                                print(f"My LLM Response text d : {text}")

                        elif 'content' in chunk:
                            text = chunk.get('content', '') or ''
                            print('\n')
                            print('\r')
                            print(f"My LLM Response text content :
{text}")

                        else:
                            # fallback - join any string values
                            pieces = [v for v in chunk.values() if
isinstance(v, str)]
                            text = " ".join(pieces)
                            print('\n')
                            print('\r')
                            print(f"My LLM Response pieces : {pieces}")

                            print('\n')
                            print('\r')
```

```
                            print(f"My LLM Response pieces text :
{text}")

                    else:
                        text = str(chunk)
                        print('\n')
                        print('\r')
                        print(f"My LLM Response final text : {text}")


                    if text:
                        response_text += text

                    print('\n')
                    print('\r')
                    print(f"My LLM Response response_text :
{response_text}")

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)

            print('\n')
            print('\r')
            print(f"My LLM Response time : {my_llm_response_time}")

            response_text = response_text.replace("# %%","")
            response_for_return = response_text
            print(f"[DEBUG AI ASSISTANT] My LLM response_for_return :
{response_for_return}")
            my_response = response_for_return
            print(f"[DEBUG AI ASSISTANT] My LLM my_response :
{my_response}")

            # --- robust, minimal extractor: prefer message.content /
content / text / choices ---
            def extract_assistant_content(obj):
                """Return assistant text only, avoid returning metadata
fields."""
                if obj is None:
                    return ""
                # direct string
                if isinstance(obj, str):
                    return obj.strip()
                # dict handling (common)
                if isinstance(obj, dict):
                    # prefer 'message' -> content
                    if 'message' in obj and obj['message']:
                        return extract_assistant_content(obj['message'])
                    # direct content/text
                    if 'content' in obj and isinstance(obj['content'],
str) and obj['content'].strip():
                        return obj['content'].strip()
                    if 'text' in obj and isinstance(obj['text'], str) and
obj['text'].strip():
```

```
                        return obj['text'].strip()
                    # choices (list of responses)
                    if 'choices' in obj and isinstance(obj['choices'],
(list, tuple)):
                        parts = [extract_assistant_content(c) for c in
obj['choices']]
                        return " ".join([p for p in parts if p])
                    # some streaming chunks use 'delta' or nested shapes
                    for key in ('delta',):
                        if key in obj and obj[key]:
                            candidate =
extract_assistant_content(obj[key])
                            if candidate:
                                return candidate
                    # fallback: scan values but skip obvious metadata
keys
                    skip_keys =
{'model','created_at','done','done_reason','total_duration','load_duratio
n',

'prompt_eval_count','prompt_eval_duration','eval_count','eval_duration'}
                    for k, v in obj.items():
                        if k in skip_keys:
                            continue
                        res = extract_assistant_content(v)
                        if res:
                            return res
                    return ""
                # list/tuple: join useful parts
                if isinstance(obj, (list, tuple)):
                    parts = [extract_assistant_content(x) for x in obj]
                    return " ".join([p for p in parts if p])
                # objects with attributes (Message-like)
                for attr in ('content', 'text', 'message', 'data'):
                    if hasattr(obj, attr):
                        try:
                            return extract_assistant_content(getattr(obj,
attr))
                        except Exception:
                            pass
                # fallback to str(obj) but only if it looks like
meaningful text (avoid dumping metadata)
                try:
                    s = str(obj).strip()
                    # crude filter: if it looks like a tuple of metadata
(many key-like words), return empty
                    if not s:
                        return ""
                    low = s.lower()
                    if ('model' in low and 'created_at' in low) or
('total_duration' in low and 'load_duration' in low):
                        return ""
                    return s
                except Exception:
```

```
                    return ""

            # assemble assistant-only content
            response_text = extract_assistant_content(ResponseOllama)

            # final cleanup (remove control markers you don't want)
            if response_text:
                response_text = response_text.replace("# %%", "").strip()
            else:
                # fallback: ensure we return at least something minimal
instead of full metadata
                response_text = ""

            # minimal debug: only the assistant text (no metadata dump)
            print(f"[DEBUG AI ASSISTANT] Assistant content:
{response_text[:1000]}")  # truncated for safety

            # preserve the rest of your flow
            response_for_return = response_text
            my_response = response_for_return

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)
            print(f"\n\rMy LLM Response time : {my_llm_response_time}")
            print(f"[DEBUG AI ASSISTANT] My LLM response_for_return :
{response_for_return}")
            # removed printing of `my_response` that previously included
metadata

            response_text_cleaned = re.sub(r"<think>.*?</think>", "",
response_text, flags=re.DOTALL).strip()
            response_text_cleaned =
clean_latex_markdown(response_text_cleaned)
            my_response = convert_to_superscript(response_text_cleaned)

            my_response = my_response.replace("I'm DeepSeek-R1","I'm
Alfred")
            my_response = my_response.replace("the Chinese Company
DeepSeek","the South African, Tjaart")
            my_response = my_response.replace("created by
DeepSeek","created by Tjaart")
            my_response = my_response.replace("exclusively by
DeepSeek","exclusively by Tjaart")

            print(f"\n[DEBUG AI ASSISTANT] My {AI_Assistant_Running} LLM
Response:\n{my_response}\n")

##            # after you generate assistant_response in English:
##            af_response = translate(my_response)
##            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans af_response
before : {af_response}")
##
##            my_response_af = af_response
```

```
##            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
my_response_af after : {my_response_af}")

            stop_llm = time.time()
            duration = int(stop_llm - start_llm)
            print(f"[DEBUG AI ASSISTANT] LLM Response Time: {duration}
seconds")

            current_date = datetime.datetime.now().strftime('%Y-%m-%d')
            current_time = datetime.datetime.now().strftime('%H:%M:%S')

            chat_entry = {
                "date": current_date,
                "time": current_time,
                "query": NewPromptEdited,
                "response": my_response,
                "model": AI_Assistant_Running
            }

            memory.add_to_memory(chat_entry)
            repeat.add_to_repeat(chat_entry)

            spoken_response = prepare_text_for_tts(my_response)

            spoken_response = spoken_response.replace("*","")
            spoken_response = spoken_response.replace("#","")

            Alfred_Repeat_Previous_Response = my_response

            # GUI log if available
            query_msg = (
                f"At {current_date} :  {current_time} : You Asked:
{AlfredQueryOffline} "
            )
            print(f"[DEBUG AI ASSISTANT query_msg] : {query_msg}")

            query_resp = f"At {current_date} :  {current_time} :
{AI_Assistant_Running} : {Alfred_Repeat_Previous_Response} :
'username':{username} "
            print(f"[DEBUG AI ASSISTANT query_resp] : {query_resp}")

            try:
                self.gui.log_message(query_msg)
                self.gui.log_response(query_resp)
            except NameError:
                print("[DEBUG AI ASSISTANT] GUI instance not available
for logging message.")

            listen.send_bluetooth(spoken_response)
            speech.AlfredSpeak(spoken_response)

            return my_response

        except  Exception as e:
```

```
            print(f"[DEBUG AI ASSISTANT] \n   error code for
AI_Assistant_Thinking : \n {e} \n Please check if your ollama server is
running...")
            pass

####    #-------------------------------------------------------------------
-------------#
####    #-------------------------- RAG AI ASSISTANT --------------------
-------------#
####    #-------------------------------------------------------------------
-------------#

    def My_Ollama_LLM_RAG(self, NewPromptEdited, NewPromptEditedSpeak,
AlfredQueryOffline):

        print(f"[DEBUG RAG AI] AlfredQueryOffline :
{AlfredQueryOffline}")
        print(f"[DEBUG RAG AI] NewPromptEdited : {NewPromptEdited}")
        print(f"[DEBUG RAG AI] NewPromptEditedSpeak :
{NewPromptEditedSpeak}")

        # call the method on this instance (important)
        message, username, timestamp =
self.AI_assistant_extract_text_from_query(NewPromptEdited)
        print(f"[DEBUG RAG AI] NEW  message : {message!r}")
        print(f"[DEBUG RAG AI] NEW  username: {username!r}")
        print(f"[DEBUG RAG AI] NEW  timestamp: {timestamp!r}")

        start_llm = time.time()

        #  Math Superscript Formatting
        def convert_to_superscript(text):
            superscript_map = {
                '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴',
                '5': '⁵', '6': '⁶', '7': '⁷', '8': '⁸', '9': '⁹'
            }
            text = re.sub(
                r'([a-zA-Z0-9])\^(\d+)',
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
            )
            text = re.sub(
                r'(\([^\)]+\))\^(\d+)',
                lambda m: m.group(1) + ''.join(superscript_map.get(c, c)
for c in m.group(2)),
                text
            )
            return text

        #  LaTeX & Markdown Cleaner
        def clean_latex_markdown(text):
            text = re.sub(r"\\begin{.*?}.*?\\end{.*?}", "", text,
flags=re.DOTALL)
```

```
            text = re.sub(r"\\frac{(.*?)}{(.*?)}", r"\1 / \2", text)
            text = re.sub(r"([a-zA-Z0-9])\^\{(\d+)\}", r"\1^\2", text)
            text = text.replace(r"\times", " × ")
            text = re.sub(r"\\sqrt{([^}]+)}", r"√\1", text)
            text = re.sub(r"\\boxed{(.*?)}", r"\1", text)
            text = re.sub(r"\*\*(.*?)\*\*", r"\1", text)
            text = re.sub(r"\\\(|\\\)|\\\[|\\\]", "", text)
            text = text.replace("{", "").replace("}", "").replace("\\",
""")
            text = re.sub(r"\n\s*\n", "\n", text)
            return text.strip()

        AI_Assistant_Running = self.gui.thinkbot_model.get() or "llama3"
        print(f"[DEBUG RAG AI]  AI_Assistant_Running (current):
{AI_Assistant_Running}")

        speech.AlfredSpeak(f"Ahhh... Let me have a look at this
document")

        Check_RAG = getattr(self, "rag_enabled", False)

        print(f"[DEBUG RAG AI]  Check_RAG : {Check_RAG}")

        try:
            contexts = self.retrieve_rag_context(message, top_k=3)
            if contexts:
                # join short contexts into system message or prepend to
prompt
                ctx_text = "\n\n".join([f"Source {i+1}:\n{c}" for i,c in
enumerate(contexts)])
                # Prepend to user message (or include as system
instruction)
                message = f"[RAG CONTEXT]\n{ctx_text}\n\n[END RAG
CONTEXT]\n\n{message}"
                print("[DEBUG RAG AI] Prepended RAG context to prompt.")
                print(f"[DEBUG RAG AI] message to prompt : {message}")
        except Exception as e:
            print("[DEBUG RAG AI] RAG retrieval error:", e)

        try :
            ResponseOllama = ollama.chat(
                model=AI_Assistant_Running,
                messages=[
                    {'role': 'system', 'content': 'You are a AI assistant
and always happy and uplifting. You will always be encouraging and
friendly. You will always give concise answers. You will always give the
full code in your responses?', 'temperature': '0.3', 'role': 'user',
'content': message, 'stream': False},
                ],
            )

            response_text = ""

            if isinstance(ResponseOllama, dict):
```

```
                message_data = ResponseOllama.get('message', {})
                response_text = message_data.get('content', '') if
isinstance(message_data, dict) else str(message_data)
            else:
                for chunk in ResponseOllama:
                    if isinstance(chunk, dict) and 'message' in chunk:
                        msg = chunk['message']
                        text = msg.get('content', '') if isinstance(msg,
dict) else str(msg)
                    else:
                        text = str(chunk)
                    response_text += text

            response_text_cleaned = re.sub(r"<think>.*?</think>", "",
response_text, flags=re.DOTALL).strip()
            my_response = response_text_cleaned
            print(f"[DEBUG RAG AI] \nMy {AI_Assistant_Running} LLM
Response:\n{my_response}\n")


##########################################################################
############
##########################################################################
############


####    #-------------------------------------------------------------
-------------#
####    #---------------------- EXTRACT ASSISTANT CONTENT ---------------
-------------#
####    #-------------------------------------------------------------
-------------#

            # --- robust, minimal extractor: prefer message.content /
content / text / choices ---
            def extract_assistant_content(obj):
                """Return assistant text only, avoid returning metadata
fields."""
                if obj is None:
                    return ""
                # direct string
                if isinstance(obj, str):
                    return obj.strip()
                # dict handling (common)
                if isinstance(obj, dict):
                    # prefer 'message' -> content
                    if 'message' in obj and obj['message']:
                        return extract_assistant_content(obj['message'])
                    # direct content/text
                    if 'content' in obj and isinstance(obj['content'],
str) and obj['content'].strip():
                        return obj['content'].strip()
                    if 'text' in obj and isinstance(obj['text'], str) and
obj['text'].strip():
```

```
                        return obj['text'].strip()
                    # choices (list of responses)
                    if 'choices' in obj and isinstance(obj['choices'],
(list, tuple)):
                        parts = [extract_assistant_content(c) for c in
obj['choices']]
                        return " ".join([p for p in parts if p])
                    # some streaming chunks use 'delta' or nested shapes
                    for key in ('delta',):
                        if key in obj and obj[key]:
                            candidate =
extract_assistant_content(obj[key])
                            if candidate:
                                return candidate
                    # fallback: scan values but skip obvious metadata
keys
                    skip_keys =
{'model','created_at','done','done_reason','total_duration','load_duratio
n',

'prompt_eval_count','prompt_eval_duration','eval_count','eval_duration'}
                    for k, v in obj.items():
                        if k in skip_keys:
                            continue
                        res = extract_assistant_content(v)
                        if res:
                            return res
                    return ""
                # list/tuple: join useful parts
                if isinstance(obj, (list, tuple)):
                    parts = [extract_assistant_content(x) for x in obj]
                    return " ".join([p for p in parts if p])
                # objects with attributes (Message-like)
                for attr in ('content', 'text', 'message', 'data'):
                    if hasattr(obj, attr):
                        try:
                            return extract_assistant_content(getattr(obj,
attr))
                        except Exception:
                            pass
                # fallback to str(obj) but only if it looks like
meaningful text (avoid dumping metadata)
                try:
                    s = str(obj).strip()
                    # crude filter: if it looks like a tuple of metadata
(many key-like words), return empty
                    if not s:
                        return ""
                    low = s.lower()
                    if ('model' in low and 'created_at' in low) or
('total_duration' in low and 'load_duration' in low):
                        return ""
                    return s
                except Exception:
```

```
                    return ""

            # assemble assistant-only content
            response_text = extract_assistant_content(ResponseOllama)

            # final cleanup (remove control markers you don't want)
            if response_text:
                response_text = response_text.replace("# %%", "").strip()
            else:
                # fallback: ensure we return at least something minimal
instead of full metadata
                response_text = ""

            # minimal debug: only the assistant text (no metadata dump)
            print(f"[DEBUG AI ASSISTANT] Assistant content:
{response_text[:1000]}")  # truncated for safety

            # preserve the rest of your flow
            response_for_return = response_text
            my_response = response_for_return

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)
            print(f"\n\rMy LLM Response time : {my_llm_response_time}")
            print(f"[DEBUG AI ASSISTANT] My LLM response_for_return :
{response_for_return}")
            # removed printing of `my_response` that previously included
metadata

##########################################################################
############
##########################################################################
############

########            # after you generate assistant_response in English:
########            af_response = translate(my_response)
########            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
af_response before : {af_response}")
########
########            my_response_af = af_response
########            print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
my_response_af after : {my_response_af}")

            stop_llm = time.time()
            duration = int(stop_llm - start_llm)
            print(f"[DEBUG RAG AI] LLM Response Time: {duration}
seconds")

            current_date = datetime.datetime.now().strftime('%Y-%m-%d')
            current_time = datetime.datetime.now().strftime('%H:%M:%S')

            chat_entry = {
                "date": current_date,
                "time": current_time,
```

```
                "query": NewPromptEdited,
                "response": my_response,
                "model": AI_Assistant_Running
            }

            memory.add_to_memory(chat_entry)
            repeat.add_to_repeat(chat_entry)

            spoken_response = my_response

            spoken_response = spoken_response.replace("*","")
            spoken_response = spoken_response.replace("#","")
            Alfred_Repeat_Previous_Response = my_response

            # GUI log if available
            query_msg = (
                f"[DEBUG RAG AI] At {current_date} :  {current_time} :
You Asked: {AlfredQueryOffline} "
            )

            query_resp = f"[DEBUG RAG AI] At {current_date} :
{current_time} : {AI_Assistant_Running} :
{Alfred_Repeat_Previous_Response} : 'username':{username} "

            try:
                self.gui.log_message(query_msg)
                self.gui.log_response(query_resp)
            except NameError:
                print("[DEBUG RAG AI] GUI instance not available for
logging message.")

            listen.send_bluetooth(spoken_response)
            speech.AlfredSpeak(spoken_response)

            return my_response

        except  Exception as e:
            print(f"[DEBUG RAG AI] \n   error code for
AI_Assistant_Thinking : \n {e} \n Please check if your ollama server is
running...")
            pass

####    #-------------------------------------------------------------
-------------#
####    #------------------------- CHAT AI ASSISTANT --------------------
-------------#
####    #-------------------------------------------------------------
-------------#

    def My_Ollama_LLM_Chat(self, AlfredQueryOffline):
        print("chat.")

        print(f"[DEBUG AI CHAT] AlfredQueryOffline :
{AlfredQueryOffline}")
```

```
        print(f"[DEBUG AI CHAT] NewPromptEdited : {NewPromptEdited}")
        print(f"[DEBUG AI CHAT] NewPromptEditedSpeak :
{NewPromptEditedSpeak}")

         # call the method on this instance (important)
        message, username, timestamp =
self.AI_assistant_extract_text_from_query(AlfredQueryOffline)
        print(f"[DEBUG AI CHAT] NEW  message : {message!r}")
        print(f"[DEBUG AI CHAT] NEW  username: {username!r}")
        print(f"[DEBUG AI CHAT] NEW  username: {timestamp!r}")

        speech.AlfredSpeak("chat.")
        print("[DEBUG AI CHAT] Going to Local (Chatting) LLM
(GPT).....!")

        global Alfred_Repeat_Previous_Response

        start_llm = time.time()

        while True:
            query = listen.listen()
            AlfredQueryOffline = query

            print(f"[DEBUG AI CHAT]  You said: {query}")
            print(f"[DEBUG AI CHAT] You said AlfredQueryOffline Chat:
{AlfredQueryOffline}")

            if not AlfredQueryOffline:
                continue

            AlfredQueryOffline_Doing = AlfredQueryOffline
            print(f"[DEBUG AI CHAT] You said AlfredQueryOffline_Doing :
{AlfredQueryOffline_Doing}")

            Alfred_No_Ollama = 1

            if 'stop' and 'chat' in AlfredQueryOffline.lower():
                speech.AlfredSpeak("Stopping chat.")
                break

            try:
                if len(AlfredQueryOffline) >= 5:
                    speech.AlfredSpeak('sure sir')

                    current_date = datetime.datetime.now().strftime('%Y-
%m-%d')
                    current_time =
datetime.datetime.now().strftime('%H:%M:%S')
                    query_msg = f"{current_date} : {current_time} :
{AlfredQueryOffline}"

                    if self.gui is not None:
                        self.gui.log_query(query_msg)
```

```
                    AI_Chat_Assistant_Running =
self.gui.chatbot_model.get() or "llama3-chatqa:8b"
                    print(f"[DEBUG AI CHAT]  AI_Chat_Assistant_Running
(current): {AI_Chat_Assistant_Running}")

                    response_text = ""

                    ResponseOllama = ollama.chat(

                        model = AI_Chat_Assistant_Running,

                        messages=[
                            {'role': 'system', 'content': 'You are a Chat
AI assistant and always happy and uplifting. You will always be
encouraging and friendly. You will always give concise answers. You will
always keep the answers sweet and short.', 'temperature': '0.3', 'role':
'user', 'content': message, 'stream': True}, ],
                        )

                    if isinstance(ResponseOllama, dict):
                        message_data = ResponseOllama.get('message', {})
                        response_text = message_data.get('content', '')
if isinstance(message_data, dict) else str(message_data)
                    else:
                        for chunk in ResponseOllama:
                            if isinstance(chunk, dict) and 'message' in
chunk:
                                msg = chunk['message']
                                text = msg.get('content', '') if
isinstance(msg, dict) else str(msg)
                            else:
                                text = str(chunk)
                            response_text += text

                    my_response = response_text
##
##                    # after you generate assistant_response in English:
##                    af_response = translate(my_response)
##                    print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
af_response before : {af_response}")
##
##                    my_response_af = af_response
##                    print(f"[DEBUG AI ASSISTANT] My LLM Afrikaans
my_response_af after : {my_response_af}")

                    print(f"[DEBUG AI CHAT] \n\rMy
{AI_Chat_Assistant_Running} LLM Response: {my_response}")
                    stop_llm = time.time()
                    my_llm_response_time = int(stop_llm - start_llm)

                    print(f"[DEBUG AI CHAT] \n\rMy LLM Response time :
{my_llm_response_time}")
```

```
print('_____
__')

                    chat_entry = {
                        "date": current_date,
                        "time": current_time,
                        "query": AlfredQueryOffline,
                        "response": my_response
                    }

                    memory.add_to_memory(chat_entry)
                    repeat.add_to_repeat(chat_entry)

                    Alfred_Repeat_Previous_Response = my_response

                    # GUI log if available
                    log_msg = (
                        f"At {current_date} :  {current_time} : You
Asked: {AlfredQueryOffline} "
                    )

                    query_resp = f"At {current_date} :  {current_time} :
{Alfred_Repeat_Previous_Response} : 'username':{username} "

                    try:
                        self.gui.log_message(log_msg)
                        self.gui.log_response(query_resp)
                    except NameError:
                        print("[DEBUG AI CHAT] GUI instance not available
for logging message.")

                    listen.send_bluetooth(my_response)
                    speech.AlfredSpeak(my_response)

                    print('[DEBUG AI CHAT] return 4')
                    Alfred_No_Ollama = 0

            except Exception as e:
                print(f"[DEBUG AI CHAT] Exception for AI_Assistant_Chat :
\n {e} \n Please check if your ollama server is running...")
                continue

######    #-------------------------------------------------------------
--------------#
######    #------------------------ CODER AI ASSISTANT ------------------
---------------#
######    #-------------------------------------------------------------
---------------#

    # --- Updated method with formatting integration (fixed streaming
parsing) ---
    def My_Ollama_LLM_Coder(self, NewPromptEdited, NewPromptEditedSpeak,
AlfredQueryOffline):
```

```
        speech.AlfredSpeak(f"Coding is fun, let's do it...")

        print(f"[DEBUG AI CODER] AlfredQueryOffline :
{AlfredQueryOffline}")
        print(f"[DEBUG AI CODER] NewPromptEdited : {NewPromptEdited}")
        print(f"[DEBUG AI CODER] NewPromptEditedSpeak :
{NewPromptEditedSpeak}")

        # call the method on this instance (important)
        message, username, timestamp =
self.AI_assistant_extract_text_from_query(NewPromptEdited)

        print(f"[DEBUG AI CODER] NEW  message : {message!r}")
        print(f"[DEBUG AI CODER] NEW  username: {username!r}")
        print(f"[DEBUG AI CODER] NEW  timestamp: {timestamp!r}")

        global Alfred_Repeat_Previous_Response

        start_llm = time.time()

        print('\r')
        print("[DEBUG AI CODER] Diagnostic Testing.....LLM Running...")
        print('\n')
        print('\r')

        print("[DEBUG AI CODER] Going to Local (Coding) LLM (GPT).....!")

        AI_Coding_Assistant_Running = self.gui.coding_model.get() or
"deepseek-coder"
        print(f"[DEBUG AI CODER]  AI_Assistant_Running (current):
{AI_Coding_Assistant_Running}")
        response_queue = queue.Queue()  # for thread communication

        print(f"[DEBUG AI CODER] Right, Let me think about
{NewPromptEditedSpeak} with {AI_Coding_Assistant_Running}")
        print(f"[DEBUG AI CODER]  AI_Coding_Assistant_Running:
{AI_Coding_Assistant_Running}")

        try:
            # build messages properly (system + user) so the system
prompt isn't overwritten
            messages = [
                {
                    'role': 'system',
                    'content': (
                        'Strictly adhere to the following rules except if
the user instruct otherwise:-'
                        '- You are an Expert Coding assistant. '
                        '- Output the raw code directly first and then
the rest.'
                        '- You will always give the full code in your
responses.'
                        '- Only generate the code requested.'
```

```
                        '- never remove any of the existing code.'
                        '- only fix what you must.'
                        '- Do not include any conversational text, keep
explanations and comments short.'
                    )
                },
                {'role': 'user', 'content': message}
            ]

            # initialize response_text for both streaming and non-
streaming cases
            response_text = ""

            ResponseOllama = ollama.chat(
                model=AI_Coding_Assistant_Running,
                messages=messages,
                stream=True,   # stream True to receive chunks
            )

            print(f"[DEBUG AI CODER] ResponseOllama : {ResponseOllama}")

            response_text = ""

            # ResponseOllama was returned by ollama.chat(...)
            if isinstance(ResponseOllama, dict):
                # non-streaming shape
                message_data = ResponseOllama.get('message', {})
                if isinstance(message_data, dict):
                    response_text = message_data.get('content', '') or ''
                else:
                    response_text = str(message_data or '')
            else:
                # streaming: assemble every chunk into one string
                import re  # local import to keep changes minimal and
safe
                for chunk in ResponseOllama:
                    text = ""

                    # 1) dict-like chunk (common)
                    if isinstance(chunk, dict):
                        if 'message' in chunk:
                            msg = chunk['message']
                            if isinstance(msg, dict):
                                text = msg.get('content', '') or ''
                            else:
                                # msg may be an object or string
                                if hasattr(msg, 'get'):
                                    text = msg.get('content', '') or ''
                                elif hasattr(msg, 'content'):
                                    text = getattr(msg, 'content') or ''
                                else:
                                    text = str(msg)
                        elif 'delta' in chunk:
                            d = chunk['delta']
```

```
                            if isinstance(d, dict):
                                text = d.get('content', '') or
d.get('text', '') or ''
                            else:
                                text = str(d)
                        elif 'content' in chunk:
                            text = chunk.get('content', '') or ''
                        else:
                            pieces = [v for v in chunk.values() if
isinstance(v, str)]
                            text = " ".join(pieces)
                    else:
                        # 2) object-like chunk: try to extract common
attributes
                        # some clients return objects with .message
(which itself may be dict/object)
                        if hasattr(chunk, 'message'):
                            msg = getattr(chunk, 'message')
                            if isinstance(msg, dict):
                                text = msg.get('content', '') or ''
                            else:
                                if hasattr(msg, 'get'):
                                    text = msg.get('content', '') or ''
                                elif hasattr(msg, 'content'):
                                    text = getattr(msg, 'content') or ''
                                else:
                                    text = str(msg)
                        # direct content attribute
                        elif hasattr(chunk, 'content'):
                            text = getattr(chunk, 'content') or ''
                        # direct delta attribute
                        elif hasattr(chunk, 'delta'):
                            d = getattr(chunk, 'delta')
                            if isinstance(d, dict):
                                text = d.get('content', '') or
d.get('text', '') or ''
                            else:
                                text = str(d)
                        # dict-like access fallback
                        elif hasattr(chunk, 'get'):
                            try:
                                text = chunk.get('content', '') or
chunk.get('text', '') or ''
                            except Exception:
                                text = str(chunk)
                        else:
                            # final fallback: try to regex-extract
"content='...'" from repr
                            s = str(chunk)
                            m = re.search(r"content='(.*?)'", s)
                            if m:
                                text = m.group(1)
                            else:
                                # last resort: keep the string conversion
```

```
                                text = s

                        if text:
                            response_text += text

            stop_llm = time.time()
            my_llm_response_time = int(stop_llm - start_llm)

            print('\n')
            print('\r')
            print(f"[DEBUG AI CODER] My LLM Response time :
{my_llm_response_time}")

            response_for_return = response_text
            print(f"[DEBUG AI CODER] My LLM response_for_return :
{response_for_return}")
            my_response = response_for_return
            print(f"[DEBUG AI CODER] My LLM my_response : {my_response}")


print('_____________________________________________________________________
__')

            time_format = time.strftime("%H:%M:%S",
time.gmtime(my_llm_response_time))
            print('\n')
            print('\r')
            print("[DEBUG AI CODER] It took " + time_format + " for the
response")
            print('\n')

print('_____________________________________________________________________
__')
            print('\n')

            current_date = datetime.datetime.now().strftime('%Y-%m-%d')
            current_time = datetime.datetime.now().strftime('%H:%M:%S')

            # Store formatted Markdown in memory (you can switch to
plain_text if preferred)
            chat_entry = {"date": current_date, "time": current_time,
"query": NewPromptEdited, "response": my_response}

            memory.add_to_memory(chat_entry)  # □ Store in memory
            repeat.add_to_repeat(chat_entry)  # □ Store last response

            Alfred_Repeat_Previous_Response = response_for_return
            print(f"[DEBUG AI CODER] Alfred_Repeat_Previous_Response :
{Alfred_Repeat_Previous_Response}")

            Did_History = Alfred_Repeat_Previous_Response
            print('[DEBUG AI CODER] Did_History : ', Did_History)
```

```
            # GUI log if available (log the plain_text for readability in
narrow UI)
            log_msg = f"At {current_date} :  {current_time} : You Asked:
{AlfredQueryOffline} "
            query_resp = f"At {current_date} :  {current_time} :
{AI_Coding_Assistant_Running} : {Alfred_Repeat_Previous_Response} :
'username':{username} "

            try:
                self.gui.log_message(log_msg)
                self.gui.log_response(query_resp)
            except NameError:
                print("[DEBUG AI CODER] GUI instance not available for
logging message in CODING LLM.")

            # Sanitize plaintext for bluetooth/speech
            to_speak = response_for_return.replace("*", "").replace("#",
"")

            listen.send_bluetooth(to_speak)
            speech.AlfredSpeak(to_speak)

            # small delay to allow speech to play out
            time.sleep(8)

            print('return 4')

            Alfred_No_Ollama = 0
            print("Alfred_No_Ollama : ", Alfred_No_Ollama)

            # Example attempt to pop TODO list (kept as you had it)
            try:
                if MyToDoListEdited != []:
                    MyToDoListEdited.pop(0)
                    AlfredQueryOfflineNew = MyToDoListEdited
                    print("")
                    print("AlfredQueryOffline New : " +
str(AlfredQueryOfflineNew))
                    print('')
            except Exception:
                pass

            return response_for_return

        except Exception as e:
            print(f"[DEBUG AI CODER] error code for AI_Assistant_chat :
\n {e} \n Please check if your ollama server is running...")
            return ""  # return empty string on error

    def show_history(self):
        """Return saved conversation history."""
        return memory.get_history()

    def reset_memory(self):
```

```
        """Clear stored memory."""
        memory.clear_memory()

print("AI_Assistant ended")

ai_assistant = AI_Assistant
```