LitellmModel

Let's dive into the `LitellmModel` class code provided, breaking it down line by line as if preparing for an exam. The goal is to explain each part clearly, covering its purpose and functionality, so you can understand and recall it effectively. This class is part of a system that integrates with LiteLLM, a library that provides a unified interface to interact with various language models (e.g., OpenAI, Anthropic, Gemini, Mistral). The class enables making API calls to these models and handling their responses, both for regular and streaming requests.

Imports and Setup

```
from __future__ import annotations
```

- **Purpose**: Enables postponed evaluation of type annotations, allowing forward references in type hints (e.g., referencing a class or type before it's defined). This is useful for modern Python type checking.
- **Explanation**: This line ensures compatibility with type hints used throughout the code, especially in complex class relationships or recursive types.

```
import json
import time
from collections.abc import AsyncIterator
from typing import Any, Literal, cast, overload
import litellm.types
from agents.exceptions import ModelBehaviorError
```

- **Purpose**: Imports standard Python libraries and specific types for type hints.
  - `json`: For serializing/deserializing JSON data, used for logging and debugging.
  - `time`: For timestamping responses (e.g., `created_at` in the `Response` object).
  - `AsyncIterator`: Used for streaming responses, which are asynchronous iterators.
  - `Any, Literal, cast, overload`: Type hint utilities from the `typing` module for flexible and precise type definitions.
  - `litellm.types`: Imports type definitions from the LiteLLM library, such as `Message` and `ModelResponse`.
  - `ModelBehaviorError`: A custom exception for handling unexpected model behavior.
- **Explanation**: These imports set up the necessary tools for handling asynchronous operations, type safety, and LiteLLM-specific types, ensuring robust error handling and data manipulation.

```
try:
    import litellm
except ImportError as _e:
    raise ImportError(
        "`litellm` is required to use the LitellmModel. You can install it via the optional "
        "dependency group: `pip install 'openai-agents[litellm]'`."
    ) from _e
```

- **Purpose**: Ensures the `litellm` library is installed, raising a helpful error if it's missing.
- **Explanation**: The `try-except` block checks if `litellm` is available. If not, it raises an `ImportError` with instructions to install the `openai-agents` package with the `litellm` extra, preserving the original exception for debugging.

```
from openai import NOT_GIVEN, AsyncStream, NotGiven
from openai.types.chat import ChatCompletionChunk, ChatCompletionMessageToolCall
from openai.types.chat.chat_completion_message import (
    Annotation,
    AnnotationURLCitation,
    ChatCompletionMessage,
)
from openai.types.responses import Response
```

- **Purpose**: Imports types from the `openai` library for compatibility with OpenAI's API response formats.
  - `NOT_GIVEN` and `NotGiven`: Represent optional parameters not provided by the user.
  - `AsyncStream`: For handling streaming responses asynchronously.
  - `ChatCompletionChunk`, `ChatCompletionMessageToolCall`, `ChatCompletionMessage`, `Annotation`, `AnnotationURLCitation`: Types for structuring chat completions, tool calls, and annotations.
  - `Response`: A custom response type for wrapping model outputs.
- **Explanation**: These imports ensure compatibility with OpenAI's data structures, as LiteLLM often maps other models' responses to OpenAI's format.

```
from ... import _debug
from ...agent_output import AgentOutputSchemaBase
from ...handoffs import Handoff
from ...items import ModelResponse, TResponseInputItem, TResponseStreamEvent
from ...logger import logger
from ...model_settings import ModelSettings
from ...models.chatcmpl_converter import Converter
from ...models.chatcmpl_helpers import HEADERS
from ...models.chatcmpl_stream_handler import ChatCmplStreamHandler
from ...models.fake_id import FAKE_RESPONSES_ID
from ...models.interface import Model, ModelTracing
from ...tool import Tool
from ...tracing import generation_span
from ...tracing.span_data import GenerationSpanData
from ...tracing.spans import Span
from ...usage import Usage
```

- **Purpose**: Imports internal modules and classes from the project's package.
  - `_debug` : Debugging utilities (e.g., `DONT_LOG_MODEL_DATA` flag).
  - `AgentOutputSchemaBase` : Base class for defining output schemas.
  - `Handoff` : Represents handoff mechanisms (e.g., passing tasks to other agents or tools).
  - `ModelResponse` , `TResponseInputItem` , `TResponseStreamEvent` : Types for model responses, input items, and streaming events.
  - `logger` : Custom logging utility for debugging and monitoring.
  - `ModelSettings` : Configuration for model parameters (e.g., temperature, max tokens).
  - `Converter` : Utility for converting between LiteLLM and OpenAI formats.
  - `HEADERS` : Predefined HTTP headers for API requests.
  - `ChatCmplStreamHandler` : Handles streaming responses.
  - `FAKE_RESPONSES_ID` : A placeholder ID for responses.
  - `Model` , `ModelTracing` , `generation_span` , `Span` , `GenerationSpanData` : Tracing utilities for monitoring model execution.
  - `Tool` : Represents tools the model can call.
  - `Usage` : Tracks token usage (input, output, total).
- **Explanation**: These imports connect the `LitellmModel` class to the broader system, providing utilities for configuration, response handling, tracing, and logging.

`LitellmModel` Class

```
class LitellmModel(Model):
    """This class enables using any model via LiteLLM. LiteLLM allows you to acess OpenAPI,
    Anthropic, Gemini, Mistral, and many other models.
    See supported models here: [litellm models](https://docs.litellm.ai/docs/providers).
    """
```

- **Purpose**: Defines the `LitellmModel` class, which inherits from a base `Model` class, to interact with various language models via LiteLLM.
- **Explanation**: The docstring explains that this class acts as a bridge to access multiple language models through LiteLLM's unified API. The link to LiteLLM's documentation indicates supported models.

```
def __init__(
    self,
    model: str,
    base_url: str | None = None,
    api_key: str | None = None,
):
    self.model = model
    self.base_url = base_url
    self.api_key = api_key
```

- **Purpose**: Initializes the `LitellmModel` instance with model-specific settings.
- **Explanation**:
  - `model: str` : Specifies the model name (e.g., "gpt-3.5-turbo", "claude-3-opus").
  - `base_url: str | None` : Optional base URL for the model's API endpoint (e.g., for custom or self-hosted models).
  - `api_key: str | None` : Optional API key for authentication.
  - These parameters are stored as instance attributes for use in API calls.

`get_response` Method

```
async def get_response(
    self,
    system_instructions: str | None,
    input: str | list[TResponseInputItem],
    model_settings: ModelSettings,
    tools: list[Tool],
    output_schema: AgentOutputSchemaBase | None,
    handoffs: list[Handoff],
    tracing: ModelTracing,
    previous_response_id: str | None,
) -> ModelResponse:
```

- **Purpose**: Asynchronously fetches a non-streaming response from the model.
- **Explanation**:
  - **Parameters**:
    - `system_instructions` : Optional system prompt to guide the model.
    - `input` : Input data, either a string or a list of `TResponseInputItem` objects (e.g., messages).
    - `model_settings` : Configuration for model behavior (e.g., temperature, max tokens).
    - `tools` : List of tools the model can call.
    - `output_schema` : Optional schema to structure the model's output.
    - `handoffs` : List of handoff tasks for other agents/tools.
    - `tracing` : Tracing configuration for monitoring execution.
    - `previous_response_id` : Optional ID of a previous response for context.
  - **Return**: A `ModelResponse` object containing the model's output, usage data, and response ID.

```
with generation_span(
    model=str(self.model),
    model_config=model_settings.to_json_dict()
    | {"base_url": str(self.base_url or ""), "model_impl": "litellm"},
    disabled=tracing.is_disabled(),
) as span_generation:
```

- **Purpose**: Creates a tracing span to monitor the model's execution.
- **Explanation**:
  - `generation_span` : A context manager that tracks the model call, including the model name, configuration, and base URL.
  - `model_settings.to_json_dict() | {"base_url": ..., "model_impl": "litellm"}` : Combines model settings with additional metadata using the `|` operator (dictionary merge, Python 3.9+).
  - `disabled=tracing.is_disabled()` : Disables tracing if specified.
  - The `span_generation` object is used to log input/output data and usage.

```
response = await self._fetch_response(
    system_instructions,
    input,
    model_settings,
    tools,
    output_schema,
    handoffs,
    span_generation,
    tracing,
    stream=False,
)
```

- **Purpose**: Calls the internal `_fetch_response` method to get the model's response.
- **Explanation**: Passes all parameters to `_fetch_response` , setting `stream=False` for a non-streaming response. The `await` keyword indicates an asynchronous call.

```
assert isinstance(response.choices[0], litellm.types.utils.Choices)
```

- **Purpose**: Verifies that the response's first choice is a valid LiteLLM `Choices` object.
- **Explanation**: Ensures type safety before processing the response, as LiteLLM's response format includes a `choices` list.

```
if _debug.DONT_LOG_MODEL_DATA:
    logger.debug("Received model response")
else:
    logger.debug(
        f"LLM resp:\n{json.dumps(response.choices[0].message.model_dump(), indent=2)}\n"
    )
```

- **Purpose**: Logs the model's response for debugging.
- **Explanation**:
  - Checks `_debug.DONT_LOG_MODEL_DATA` to decide whether to log detailed response data.
  - If logging is enabled, serializes the first choice's message to JSON with indentation for readability.

```
if hasattr(response, "usage"):
    response_usage = response.usage
    usage = (
        Usage(
            requests=1,
            input_tokens=response_usage.prompt_tokens,
            output_tokens=response_usage.completion_tokens,
            total_tokens=response_usage.total_tokens,
        )
        if response.usage
        else Usage()
    )
else:
    usage = Usage()
    logger.warning("No usage information returned from Litellm")
```

- **Purpose**: Extracts token usage data from the response.
- **Explanation**:
  - Checks if the response has a `usage` attribute.
  - If present, creates a `Usage` object with request count, input/output/total tokens.
  - If absent, logs a warning and returns an empty `Usage` object.

```
if tracing.include_data():
    span_generation.span_data.output = [response.choices[0].message.model_dump()]
span_generation.span_data.usage = {
    "input_tokens": usage.input_tokens,
    "output_tokens": usage.output_tokens,
}
```

- **Purpose**: Logs response data and usage in the tracing span.
- **Explanation**:
  - If tracing includes data (`tracing.include_data()`), stores the response message (serialized via `model_dump`) in the span.
  - Records input and output token counts in the span's usage data.

```
items = Converter.message_to_output_items(
    LitellmConverter.convert_message_to_openai(response.choices[0].message)
)
```

- **Purpose**: Converts the model's response message to output items.
- **Explanation**:
  - `LitellmConverter.convert_message_to_openai` : Converts LiteLLM's message format to OpenAI's `ChatCompletionMessage` format.
  - `Converter.message_to_output_items` : Transforms the message into a list of output items compatible with the system's format.

```
return ModelResponse(
    output=items,
    usage=usage,
    response_id=None,
)
```

- **Purpose**: Returns the final response object.
- **Explanation**: Creates a `ModelResponse` with the converted output items, usage data, and a `None` response ID (as LiteLLM may not provide one).

`stream_response` Method

```
async def stream_response(
    self,
    system_instructions: str | None,
    input: str | list[TResponseInputItem],
    model_settings: ModelSettings,
    tools: list[Tool],
    output_schema: AgentOutputSchemaBase | None,
    handoffs: list[Handoff],
    tracing: ModelTracing,
    *,
    previous_response_id: str | None,
) -> AsyncIterator[TResponseStreamEvent]:
```

- **Purpose**: Asynchronously streams model responses as an iterator of events.
- **Explanation**:
  - Similar parameters to `get_response`, but returns an `AsyncIterator[TResponseStreamEvent]` for streaming.
  - The `previous_response_id` is a keyword-only argument (indicated by `*`).

```
with generation_span(...) as span_generation:
    response, stream = await self._fetch_response(
        system_instructions,
        input,
        model_settings,
        tools,
        output_schema,
        handoffs,
        span_generation,
        tracing,
        stream=True,
    )
```

- **Purpose**: Initiates a streaming response.
- **Explanation**:
  - Creates a tracing span similar to `get_response`.
  - Calls `_fetch_response` with `stream=True`, returning a tuple of a `Response` object and an `AsyncStream` of `ChatCompletionChunk` objects.

```
final_response: Response | None = None
async for chunk in ChatCmplStreamHandler.handle_stream(response, stream):
    yield chunk
    if chunk.type == "response.completed":
        final_response = chunk.response
```

- **Purpose**: Processes the streaming response.
- **Explanation**:
  - Iterates over chunks using `ChatCmplStreamHandler.handle_stream`.
  - Yields each chunk (of type `TResponseStreamEvent`) to the caller.
  - Stores the final response when a `response.completed` chunk is received.

```
if tracing.include_data() and final_response:
    span_generation.span_data.output = [final_response.model_dump()]
if final_response and final_response.usage:
    span_generation.span_data.usage = {
        "input_tokens": final_response.usage.input_tokens,
        "output_tokens": final_response.usage.output_tokens,
    }
```

- **Purpose**: Logs the final response and usage in the tracing span.
- **Explanation**: Similar to `get_response`, but only logs if a `final_response` is available and tracing is enabled.

`_fetch_response` Method

```
@overload
async def _fetch_response(
    self,
    system_instructions: str | None,
    input: str | list[TResponseInputItem],
    model_settings: ModelSettings,
    tools: list[Tool],
    output_schema: AgentOutputSchemaBase | None,
    handoffs: list[Handoff],
    span: Span[GenerationSpanData],
    tracing: ModelTracing,
    stream: Literal[True],
) -> tuple[Response, AsyncStream[ChatCompletionChunk]]: ...

@overload
async def _fetch_response(
    self,
    system_instructions: str | None,
    input: str | list[TResponseInputItem],
    model_settings: ModelSettings,
    tools: list[Tool],
    output_schema: AgentOutputSchemaBase | None,
    handoffs: list[Handoff],
    span: Span[GenerationSpanData],
    tracing: ModelTracing,
    stream: Literal[False],
) -> litellm.types.utils.ModelResponse: ...
```

- **Purpose**: Defines two overloaded signatures for `_fetch_response` to handle streaming and non-streaming cases.
- **Explanation**:
  - Uses `@overload` for type checking to specify different return types based on the `stream` parameter.
  - `stream=True`: Returns a tuple of `Response` and `AsyncStream[ChatCompletionChunk]`.
  - `stream=False`: Returns a `litellm.types.utils.ModelResponse`.

```
async def _fetch_response(
    self,
    system_instructions: str | None,
    input: str | list[TResponseInputItem],
    model_settings: ModelSettings,
    tools: list[Tool],
    output_schema: AgentOutputSchemaBase | None,
    handoffs: list[Handoff],
    span: Span[GenerationSpanData],
    tracing: ModelTracing,
    stream: bool = False,
) -> litellm.types.utils.ModelResponse | tuple[Response, AsyncStream[ChatCompletionChunk]]:
```

- **Purpose**: Implements the core logic for fetching model responses.
- **Explanation**: Combines streaming and non-streaming logic, returning different types based on the `stream` flag.

```
converted_messages = Converter.items_to_messages(input)
if system_instructions:
    converted_messages.insert(
        0,
        {
            "content": system_instructions,
            "role": "system",
        },
    )
if tracing.include_data():
    span.span_data.input = converted_messages
```

- **Purpose**: Prepares input messages for the model.
- **Explanation**:
  - Converts input (string or `TResponseInputItem` list) to a list of messages using `Converter.items_to_messages`.

    - If `system_instructions` is provided, inserts a system message at the start.
    - Logs the input messages in the tracing span if enabled.

```
parallel_tool_calls = (
    True
    if model_settings.parallel_tool_calls and tools and len(tools) > 0
    else False
    if model_settings.parallel_tool_calls is False
    else None
)
tool_choice = Converter.convert_tool_choice(model_settings.tool_choice)
response_format = Converter.convert_response_format(output_schema)
converted_tools = [Converter.tool_to_openai(tool) for tool in tools] if tools else []
for handoff in handoffs:
    converted_tools.append(Converter.convert_handoff_tool(handoff))
```

- **Purpose**: Configures model parameters and tools.
- **Explanation**:
    - `parallel_tool_calls` : Determines if multiple tool calls can be made in parallel, based on `model_settings` and tool availability.
    - `tool_choice` : Converts the tool choice setting to a format compatible with LiteLLM/OpenAI.
    - `response_format` : Converts the output schema to a response format.
    - `converted_tools` : Converts `Tool` and `Handoff` objects to OpenAI-compatible tool definitions.

```
if _debug.DONT_LOG_MODEL_DATA:
    logger.debug("Calling LLM")
else:
    logger.debug(
        f"Calling Litellm model: {self.model}\n"
        f"{json.dumps(converted_messages, indent=2)}\n"
        f"Tools:\n{json.dumps(converted_tools, indent=2)}\n"
       偶
        f"Stream: {stream}\n"
        f"Tool choice: {tool_choice}\n"
        f"Response format: {response_format}\n"
    )
```

- **Purpose**: Logs the API call details.
- **Explanation**: Similar to `get_response` , logs the model call parameters, including messages, tools, and settings, unless `DONT_LOG_MODEL_DATA` is set.

```
reasoning_effort = model_settings.reasoning.effort if model_settings.reasoning else None
stream_options = None
if stream and model_settings.include_usage is not None:
    stream_options = {"include_usage": model_settings.include_usage}
extra_kwargs = {}
if model_settings.extra_query:
    extra_kwargs["extra_query"] = model_settings.extra_query
if model_settings.metadata:
    extra_kwargs["metadata"] = model_settings.metadata
if model_settings.extra_body and isinstance(model_settings.extra_body, dict):
    extra_kwargs.update(model_settings.extra_body)
```

- **Purpose**: Configures additional API call parameters.
- **Explanation**:
    - `reasoning_effort` : Extracts reasoning effort level if specified.
    - `stream_options` : Sets streaming options, including usage tracking.
    - `extra_kwargs` : Collects additional parameters like query, metadata, or extra body data.

```
ret = await litellm.acompletion(
    model=self.model,
    messages=converted_messages,
    tools=converted_tools or None,
    temperature=model_settings.temperature,
    top_p=model_settings.top_p,
    frequency_penalty=model_settings.frequency_penalty,
    presence_penalty=model_settings.presence_penalty,
    max_tokens=model_settings.max_tokens,
    tool_choice=self._remove_not_given(tool_choice),
    response_format=self._remove_not_given(response_format),
    parallel_tool_calls=parallel_tool_calls,
    stream=stream,
    stream_options=stream_options,
    reasoning_effort=reasoning_effort,
    extra_headers={**HEADERS, **(model_settings.extra_headers or {})},
    api_key=self.api_key,
    base_url=self.base_url,
    **extra_kwargs,
)
```

- **Purpose**: Makes the asynchronous API call to LiteLLM.
- **Explanation**: Calls `litellm.acompletion` with all configured parameters, including model settings, tools, and authentication details.

```
if isinstance(ret, litellm.types.utils.ModelResponse):
    return ret
response = Response(
    id=FAKE_RESPONSES_ID,
    created_at=time.time(),
    model=self.model,
    object="response",
    output=[],
    tool_choice=cast(Literal["auto", "required", "none"], tool_choice)
    if tool_choice != NOT_GIVEN
    else "auto",
    top_p=model_settings.top_p,
    temperature=model_settings.temperature,
    tools=[],
    parallel_tool_calls=parallel_tool_calls or False,
    reasoning=model_settings.reasoning,
)
return response, ret
```

- **Purpose**: Handles the API response.
- **Explanation**:
  - For non-streaming (`isinstance(ret, ModelResponse)`), returns the response directly.
  - For streaming, creates a `Response` object with placeholder data and returns it with the stream.

```
def _remove_not_given(self, value: Any) -> Any:
    if isinstance(value, NotGiven):
        return None
    return value
```

- **Purpose**: Converts `NotGiven` values to `None` for compatibility.
- **Explanation**: Handles cases where optional parameters are not provided, ensuring they are passed as `None` to LiteLLM.

`LitellmConverter` Class

```
class LitellmConverter:
    @classmethod
    def convert_message_to_openai(
        cls, message: litellm.types.utils.Message
    ) -> ChatCompletionMessage:
```

- **Purpose**: Converts LiteLLM's message format to OpenAI's `ChatCompletionMessage`.
- **Explanation**: Ensures compatibility by mapping fields like content, tool calls, and provider-specific data (e.g., refusal, audio).

```
@classmethod
def convert_annotations_to_openai(
    cls, message: litellm.types.utils.Message
) -> list[Annotation] | None:
```

- **Purpose**: Converts LiteLLM annotations to OpenAI's `Annotation` format.
- **Explanation**: Specifically handles URL citations, mapping fields like start/end indices, URL, and title.

```
@classmethod
def convert_tool_call_to_openai(
    cls, tool_call: litellm.types.utils.ChatCompletionMessageToolCall
) -> ChatCompletionMessageToolCall:
```

- **Purpose**: Converts LiteLLM tool calls to OpenAI's tool call format.
- **Explanation**: Maps tool call ID, type, and function details (name, arguments).

Summary for Exam Preparation

- **Purpose of `LitellmModel`**: Acts as a wrapper to interact with various language models via LiteLLM, supporting both non-streaming (`get_response`) and streaming (`stream_response`) responses.
- **Key Methods**:
  - `get_response`: Fetches a complete response, processes usage, and converts output.
  - `stream_response`: Streams response chunks, handling them via `ChatCmplStreamHandler`.
  - `_fetch_response`: Core logic for API calls, handling message conversion, tool setup, and streaming/non-streaming responses.
- **Key Features**:
  - Supports system instructions, tools, and output schemas.
  - Integrates tracing for monitoring execution and usage.
  - Converts between LiteLLM and OpenAI formats using `LitellmConverter`.
- **Error Handling**: Checks for `litellm` import, validates response types, and logs warnings for missing usage data.
- **Logging and Debugging**: Controlled by `_debug.DONT_LOG_MODEL_DATA` for detailed or minimal logging.

This breakdown covers the class's structure, methods, and purpose, preparing you to explain its functionality and implementation details for your quiz. Let me know if you need further clarification or practice questions!