keys2text_proxy/api_openai.py Script for Video:

[Scene 1: code editor showing the imports section]
code: `from fastapi import Request` and `from openai import OpenAI`
Voiceover:
"We start by importing the required modules and libraries, including FastAPI
for the API framework and OpenAI for integrating with OpenAI's services."

[Scene 2: Showing the environment variable loading setup]
code: `load_dotenv()` and `api_key = os.getenv("OPENAI_API_KEY")`
Voiceover:
"Next, we load the OpenAI API key from environment variables using dotenv.
This ensures secure and flexible configuration management."

[Scene 3: Focus on the `openai_models` function]
code: `async def openai_models()`
Voiceover:
"This function retrieves a list of available models from OpenAI.
It connects to the OpenAI API client, fetches model data, and returns a
sorted list of model IDs."

[Scene 4: Zooming into utility functions]
code: `def word_count(s)` and `def extract_request_data(request_data)`
Voiceover:
"We then define utility functions to handle tasks like word counting,
extracting request data, and formatting text. These make the code modular and reusable."

[Scene 5: Demonstrating the `log_me_request` function]
code: `def log_me_request(chat_file_name, model, user_request)`
Voiceover:
"Here, we log user requests to a file for debugging and record-keeping.
The function processes user messages and formats them for logging."

[Scene 6: codeing the response logging functionality]
code: `def log_ai_response(chat_file_name, model, backend_response)`
Voiceover:
"Similarly, responses from the AI are logged using `log_ai_response`.
It ensures both user input and AI output are traceable."

[Scene 7: Explanation of error handling]
code: `def exception_to_dict(e, params_model, status_code=500, response_text=None)`
Voiceover:
"To handle errors gracefully, we have the `exception_to_dict` function,
which captures error details and formats them into a structured dictionary."

[Scene 8: Breakdown of the `chat_completion_json` function]
code: `async def chat_completion_json(request_data, chat_file)`
Voiceover:
"This function handles non-streaming chat completions. It extracts request parameters, logs the request, and makes an API call to OpenAI. If successful, it logs and returns the response. Errors are caught and logged."

[Scene 9: Detailing the `chat_completion_stream` function]
code: `async def chat_completion_stream(request_data, chat_file)`
Voiceover:
"For streaming completions, the `chat_completion_stream` function processes the request in chunks. Each chunk is logged and streamed back to the client."

[Scene 10: Wrapping up with reusable structure]
Voiceover:
"This API handler is structured to ensure clear functionality, robust error handling, and comprehensive logging, making it an essential part of the application."