keys2text_proxy/main.py Script for Video:

[Scene 1: code editor displaying the imports section]
code: `from fastapi import FastAPI, Request, Response`
Voiceover:
"We start by importing the required modules and dependencies.
Here's where the magic begins: we bring in FastAPI and other essential libraries to build our application."

[Scene 2: code editor showing the constants and variables section]
code: `timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")`
Voiceover:
"Next, we initialize constants and variables.
Notice how a timestamped chat file is created, and a dictionary is set up to map providers to their respective API handlers."

[Scene 3: zooming into utility function definitions]
code: `def datetime_to_timestamp(date):` and
`def append_models_to_all(models, provider):`
Voiceover:
"Utility functions like this one handle conversions and organize models efficiently.
They ensure our app is well-prepared to process data seamlessly."

[Scene 4: Visual of code where the lifespan manager is defined, with transitions showing different tasks]
code: `@asynccontextmanager async def lifespan(app: FastAPI):`
Voiceover:
"Here, we define an async lifespan manager for the FastAPI app.
It takes care of tasks like opening the chat file, fetching models, appending them to a global list,
and logging session details."

[Scene 5: showing the FastAPI app instantiation with middleware setup]
code: `app = FastAPI(lifespan=lifespan)`
Voiceover:
"Now, we instantiate the FastAPI app. Middleware is added to manage cross-origin requests securely."

[Scene 6: Focus on the endpoint definition for listing models]
code: `@app.get("/v1/models")`
Voiceover:
"Our first endpoint lists all available models in JSON format.
This simple and efficient endpoint is a key part of the app."

[Scene 7: Breaking down the chat completion endpoint]
code: `@app.post("/v1/chat/completions")`
Voiceover:
"The chat completion endpoint is where the real action happens.
It begins by parsing the incoming request to identify the desired model and preferences."

[Scene 8: Zoom into the handler selection and provider-to-API mapping]
code: `provider_to_api_handler = {...}`
Voiceover:
"Based on the requested model, the appropriate API handler is selected.
Here are the imported handlers, each tailored to a specific provider."
Text overlay:
- Mock handlers for testing.
- Handlers for Anthropic, Google, Groq, OpenAI, OpenRouter, Ollama, LMStudio, and DeepSeek.

[Scene 9: showing response handling for streaming or non-streaming requests]
code: `return StreamingResponse(...)` or
`return JSONResponse(...)`
Voiceover:
"Depending on the request type, the app returns either a streaming or non-streaming response. It's a dynamic and user-friendly approach."

[Scene 10: Code section showing the main function]
code: `def main():` and
`uvicorn.run("keys2text_proxy.main:app", host="127.0.0.1", port=args.port)`
Voiceover:
"Finally, we have the main function.
It uses Uvicorn to launch the app and listens for requests on the specified port."

[Scene 11: Code showing the entry point of the script]
code: `if __name__ == "__main__": main()`
Voiceover:
"And the script execution begins here.
If you run this as the main module, the app kicks off:
loading provider and model names based on your API keys."