Report on Multi-Model LLM Chat Application

# 1. Introduction

This report analyzes a Python application designed to provide a user interface for interacting with multiple Large Language Models (LLMs). The application utilizes the Gradio library for the frontend and makes API calls to DeepSeek, Kimi (Moonshot AI), and Tongyi (Alibaba Cloud) models. The analysis addresses specific questions regarding API call variations, UI design choices, the impact of LLM parameters, and recommended prompt writing styles.

# 2. Analysis Based on Provided Questions

## 2.1. Differences in API Calls for Different Models

Yes, there are distinct differences in how the API calls are structured and handled for the three different models (DeepSeek, Kimi, Tongyi) integrated into the provided code. These differences are primarily managed within the call\_llm\_api function and the MODEL\_ENDPOINTS dictionary.

The key differences observed in the code are:

### 2.1.1. API Endpoints:

Each model uses a completely different URL endpoint for its chat completion API.

* + DeepSeek: https://api.deepseek.com/v1/chat/completions
  + Kimi: https://api.moonshot.cn/v1/chat/completions
  + Tongyi: https://dashscope.aliyuncs.com/api/v1/services/aigc/text-generation/generation

### 2.1.2. Authorization Headers:

While all use Bearer token authentication, each requires a distinct API key stored in the API\_KEYS dictionary and formatted into the Authorization header specific to its endpoint.

### 2.1.3. Request Payload Structure (data):

The JSON payload sent in the POST request varies significantly, especially for Tongyi:

DeepSeek & Kimi: Follow a structure similar to the OpenAI standard, where messages (a list of user/assistant dictionaries) and parameters like temperature are top-level keys within the JSON object. They also require a model key specifying the exact model variant (e.g., "deepseek-chat", "moonshot-v1-8k").

Tongyi: Uses a different structure. The model identifier ("qwen-turbo") is a top-level key. The conversation history (messages) is nested within an input object ("input": {"messages": [...]}). Parameters like temperature are nested within a parameters object ("parameters": {"temperature": ...}).

### 2.1.4. Response Parsing:

The structure of the successful JSON response from each API differs, requiring different parsing logic to extract the model's reply:

* + **DeepSeek & Kimi:** The assistant's message content is located within response.json()["choices"][0]["message"]["content"].
  + **Tongyi:** The assistant's message content is found at response.json()["output"]["text"].

### 2.1.5. Default Parameters:

Although temperature is the only parameter shown, the code sets different default values for each model (deepseek: 0.7, kimi: 0.3, tongyi: 0.5), indicating potential model-specific tuning or different API defaults.

These variations necessitate conditional logic (if/elif/else blocks) within the call\_llm\_api function to correctly format the request and parse the response for each selected model.

## 2.2. User Interface (UI) Design Rationale and Component Roles

The UI design, implemented using Gradio, follows a common and intuitive pattern for chat applications, aiming for usability and clarity, especially when managing multiple models and conversation histories.

### 2.2.1. Design Rationale:

Familiarity: The layout adopts a standard chat interface with a main chat window and a sidebar for history, which users are generally accustomed to.

Separation of Concerns: The UI clearly separates the conversation history management (sidebar) from the active chat interaction (main area) and model selection/input controls.

Multi-Model Accessibility: Placing the model selector (gr.Dropdown) directly above the send button makes it easy for the user to switch between LLMs for different turns within the same conversation or across different chats.

State Management: The use of gr.State for current\_session and history\_sessions allows the application to maintain conversation context and history lists without relying solely on UI components for data storage.

Visual Feedback: Custom CSS is used to enhance readability and user experience, such as differentiating user and bot messages (user-bubble, bot-bubble) and styling buttons.

### 2.2.2. Component Roles:

gr.Blocks: The main container that organizes the overall layout of the application.

gr.Markdown: Used for displaying titles ("🚀 LLM 多模型对话系统", "### 📚 历史会话").

gr.Row / gr.Column: Structure the layout into a sidebar (scale=2) and a main chat area (scale=8).

Sidebar Column (scale=2):

gr.Radio (history\_list): Displays titles (timestamps) of past conversations. Allows users to select and load a previous chat session. Its choices are dynamically updated via history\_sessions.change.

gr.Button ("🆕 新建聊天"): Clears the current chat display and internal state to start a fresh conversation session.

Main Chat Column (scale=8):

gr.Chatbot (chatbot): The primary display area showing the back-and-forth messages between the user and the selected LLM, using avatars for differentiation.

gr.Textbox (msg): The input field where the user types their messages. Configured for multi-line input and placeholder text.

gr.Dropdown (model\_selector): Allows the user to choose which LLM (DeepSeek, Kimi, Tongyi) will process the next message sent.

gr.Button ("📤 发送"): Submits the text from the msg textbox to the respond function for processing by the selected LLM.

gr.Button ("🗑️ 清空输入"): Clears the content of the msg textbox.

gr.State (current\_session, history\_sessions): Non-visible components holding the application's state: the data for the currently active chat session (ID, title, messages) and the list containing all past session data.

This design prioritizes a clean interaction flow, enabling users to easily chat, switch models, and manage their conversation history.

## 2.3. Impact of LLM Parameters (Temperature, Nucleus Sampling, etc.)

Large Language Models often expose several parameters that allow users to influence the characteristics of the generated output. The provided code explicitly utilizes the temperature parameter.

### 2.3.1. Temperature:

Function: Controls the randomness of the output. Technically, it adjusts the probabilities of potential next words before sampling.

Impact:

Low Temperature (e.g., < 0.5, like the 0.3 for Kimi): Makes the model more deterministic and focused. It tends to pick the highest probability words, resulting in outputs that are more predictable, coherent, and often conservative or repetitive. It's useful for tasks requiring factual accuracy or constrained answers.

High Temperature (e.g., > 0.7, like the 0.7 for DeepSeek): Increases randomness. The model is more likely to explore less probable words, leading to more diverse, creative, surprising, or even potentially nonsensical or off-topic outputs. It's suitable for creative writing, brainstorming, or generating varied options.

In this code: Different default temperatures (0.7, 0.3, 0.5) are set for DeepSeek, Kimi, and Tongyi respectively, suggesting a deliberate choice to elicit slightly different response styles from each model by default.

### 2.3.2. Nucleus Sampling (Top-p):

(Not explicitly used in the provided code, but a common related parameter)

Function: An alternative method to control randomness. Instead of considering all words, the model considers only the smallest set of most probable words whose cumulative probability exceeds the threshold p.

Impact:

Low Top-p (e.g., 0.1): Very restrictive, similar to very low temperature, focusing only on the most likely words.

High Top-p (e.g., 0.9): Allows for more diversity by including a wider range of probable words, but dynamically truncates the long tail of very low-probability (often irrelevant) words, which can sometimes be an advantage over high temperature.

Relationship to Temperature: Temperature and Top-p can sometimes be used together, but often one is chosen as the primary method to control output randomness. Setting top\_p to 1 effectively disables it.

## 2.4. Recommended Prompt Styles for Daily vs. Professional Use

The way a prompt (the input query or instruction given to the LLM) is written heavily influences the quality and relevance of the model's output. Different styles are suitable for different contexts.

### 2.4.1. Daily Use / Casual Interaction:

Characteristics: Often involves seeking information, simple tasks, or conversation. Prompts tend to be shorter, more informal, and resemble natural human conversation. Less emphasis is placed on structuring the output.

Recommended Style:

Natural Language: Just ask the question directly (e.g., "What's the capital of France?", "Tell me a short story about a brave knight").

Simplicity: Keep prompts concise and to the point.

Implicit Context: Rely more on the LLM's general knowledge base.

Example: "What are some good recipes for leftover chicken?"

### 2.4.2. Professional / Specialized Use:

Characteristics: Often involves complex tasks, specific knowledge domains, generating structured content, coding, analysis, or adopting a particular persona. Requires more precision and control over the output.

Recommended Style:

Clarity and Specificity: Clearly state the objective, desired format, constraints, and any necessary context. Avoid ambiguity.

Role Prompting: Instruct the model to act as a specific expert or persona (e.g., "Act as a senior data analyst...", "You are a helpful travel agent...").

Structured Instructions: Use bullet points or numbered lists for complex instructions.

Context Provision: Include relevant background information, data snippets, or previous parts of the conversation explicitly within the prompt.

Output Formatting: Specify the desired output format (e.g., "Provide the answer as a JSON object", "Write a summary in three bullet points", "Generate Python code").

Constraints: Define limitations like length ("in under 100 words"), tone ("use a formal tone"), or content ("do not include technical jargon").

Few-Shot Prompting: Provide examples of the desired input/output format within the prompt to guide the model.

Example: "Act as a marketing consultant. Analyze the following product description [insert description] and provide 3 suggestions for improving its appeal to millennial consumers. Format your suggestions as a numbered list, with a brief explanation for each."

# 3. Conclusion

The provided Python code successfully implements a functional multi-model LLM chat interface using Gradio. It correctly handles the necessary variations in API calls for DeepSeek, Kimi, and Tongyi. The UI design is user-friendly and facilitates model comparison and history management. Understanding the impact of LLM parameters like temperature and employing appropriate prompt engineering techniques—ranging from simple conversational prompts for daily use to structured, detailed prompts for professional tasks—are crucial for leveraging the full potential of these powerful language models, as enabled by this application.