Technical Report: Multi-LLM AI Support Automation via OpenRouter

Paulo Ricardo Protachevicz

1 Proposed Multi-Model Architecture

To automate customer service with generative AI in a cost-effective and high-performance way, I propose a multi-LLM orchestration architecture with the following main components:

- **Intelligent Routing Layer** – A router analyzes each question and decides which language model to use, according to criteria such as query complexity, importance, cost per token, and expected latency. Simple queries or FAQs can be handled by smaller/cheaper models, while complex or critical questions are routed to more powerful models (e.g., GPT-4). This dynamic routing ensures that trivial tasks use low-cost LLMs, reserving advanced models only when necessary. For example, common questions ("What are your support hours?") can use a basic model, while a complex or ambiguous query would be escalated to a top-tier model.

- **Models Layer via OpenRouter** – We use OpenRouter as a unified gateway for multiple LLM providers (OpenAI, Anthropic, Google, etc). OpenRouter facilitates access to several models through a single API, although it does not do routing itself (that is, the application actively decides which model to call for each request). The call is then made to the endpoint of the chosen model via OpenRouter (which manages authentication and request formatting in a standardized way). This design allows flexibility in switching models without changing the business logic. It also allows incorporating future models (e.g., Google Gemini) as they become available, simply by adjusting the routing rules.

- **Vectorized Knowledge Base** – A vector database stores supporting information (FAQ, conversation history, internal documents) as embeddings. Before responding, the agent can search this base for similar questions or relevant content, optimizing accuracy. For example, upon receiving a question, the system first obtains semantic vectors for the query and searches the vector database for similar FAQs or previous answers. If it finds strong matches (above a similarity threshold), it can use that answer directly or provide this context to the chosen LLM (Retrieval-Augmented Generation, or RAG). This reduces costs and latency by avoiding unnecessary calls to models for already answered or easily resolvable questions.

- **Interaction Flow and Memory** – The recent user conversation history is also considered. A brief context (e.g., last interactions) is maintained for continuity in multi-turn dialogues. This context can be temporarily stored (in cache or short-term memory) or even indexed in the vector database for retrieval. Thus, if the user follows up ("could you elaborate more?"), the system provides the model both the current question and the last relevant messages. This memory buffer (like in n8n flows, which use a memory buffer of the last 10 messages) ensures that the bot responds contextually.

- **Monitoring and Guardrails Module** – In addition to routing by complexity, the architecture can include guardrails and monitoring: for example, monthly spend limits per model, fallback policies if a provider fails, or security filters (PII redaction, prompt injection prevention, etc). Simple checks can also be implemented, like "if this month's cost exceeds X, downgrade all calls to a cheaper model." Usage metrics, response times, and quality are also logged to refine routing over time.

2 How the Flow Works (Overview)

1. The user sends a question (via website chat, Slack, email, etc).
2. The recent history and metadata of the question are gathered. Optionally, the system searches the knowledge vector for a similar existing answer. If a very high match is found (e.g., the same question was asked before), it can return that answer immediately (or with minimal formatting), saving time and tokens. Otherwise, it proceeds.
3. The LLM Router evaluates the question. It can use deterministic rules (e.g., question length, presence of certain technical keywords) and/or a trained classifier to estimate complexity. Based on this, it chooses an appropriate model – for example, GPT-3.5 for direct and short questions, Anthropic Claude for extensive analytical questions, or even a local open-source model for simple low-risk queries.
4. The question (enriched with relevant context, if any) is sent to the selected model via OpenRouter. OpenRouter forwards it to the corresponding provider endpoint.
5. The model generates and returns the answer. The system can post-process the answer – for example, apply Markdown formatting, insert relevant references (if it used RAG with documents), or adjust the tone according to company guidelines.
6. The interaction (customer question + given answer + model used + timestamp) is then stored in the vector database as a new knowledge item. We store the embedding of the question and possibly of the answer. This allows for quick future searches if this question (or a similar one) reappears, and maintains a semantic log of the support.
7. The final answer is sent to the user through the same original channel (chat, Slack, email). If it is a Slack case, the bot posts the answer in the thread; if via API/web, it returns to the frontend to display.

This architecture automatically balances cost and quality. Studies indicate that this LLM routing strategy can save 30-80% of token spending, maintaining similar quality. In summary, dynamic routing between models ensures that simple tasks go to cheap and fast models, while complex queries use robust models, optimizing resources.

3 Prototype Implementation (Python)

To validate the architecture, a prototype was implemented in Python (Jupyter Notebook) simulating the described flow. The code is modular and commented, prepared for future integrations. It performs: question input, dynamic routing to a simulated model, and storage of the log in a simplified "vector database."

Implementation Decisions: The code is structured in distinct functions for clarity:

- **router_model(question):** Decides which model to use based on simple rules (here, question length as a proxy for complexity). Example: questions with fewer than 10 words are routed to a fast/economical model; otherwise, to a more powerful model.
- **call_model(model_name, question):** Simulates the call to the model API via OpenRouter. Since we don't have the real API credentials in this context, we generate a fake answer indicating which model would have been used.
- **vectorize_text(text):** Generates a vector embedding for a given text. Note: In the prototype, we use a deterministic simulation (e.g., random vector with fixed seed by text hash) just to demonstrate the pipeline. In production, we would use real embeddings (OpenAI Embeddings, Sentence Transformers, etc).
- **store_interaction(question, answer, model):** Stores the interaction record. We keep a global list that represents the vector database, saving tuples with (vector, question, answer, model, timestamp).
- **search_similar(question):** (optional) Performs semantic search in the "vector database" for a new question, returning the most similar registered interaction if the cosine similarity exceeds a threshold. This demonstrates the semantic search for similar questions.

4 Code

Below is an illustrative code of the prototype:

```
import hashlib, random, math
from datetime import datetime

# Simulated vector database (list of dicts with embedding and data)
vector_db = []

def vectorize_text(text, dim=64):
"""Generates a deterministic simulated embedding for a text."""
# Use text hash as seed to generate fixed pseudo-random vector
h = int(hashlib.sha256(text.encode('utf-8')).hexdigest(), 16)
random.seed(h)
# Fixed-dimension vector with float values in [0,1)
return [random.random() for _ in range(dim)]

def router_model(question):
"""Decides which LLM to use (simple heuristic based on question length)."""
```

```
words = question.split()
if len(words) < 10:
return "openai/gpt-3.5-turbo"   # cheaper/faster model
else:
return "openai/gpt-4"           # more advanced (expensive, high-performance) model

def call_model(model, question):
"""Simulates call to selected model via OpenRouter and returns answer."""
# In a real scenario, we would use the OpenRouter API here:
# e.g., openrouter.complete(prompt=..., model=model)
return f"[Simulated response from model {model} to question: '{question}']"

def store_interaction(question, answer, model):
"""Stores question & answer in the vector database with its embedding."""
embedding = vectorize_text(question)
log_entry = {
"question": question,
"answer": answer,
"model": model,
"embedding": embedding,
"timestamp": datetime.now()
}
vector_db.append(log_entry)

def cosine(v1, v2):
dot = sum(a*b for a, b in zip(v1, v2))
norm1 = math.sqrt(sum(a*a for a in v1))
norm2 = math.sqrt(sum(b*b for b in v2))
if norm1 == 0 or norm2 == 0:
return 0.0
return dot / (norm1 * norm2)

def search_similar(question, threshold=0.8):
"""Searches for a similar question in the vector database via cosine similarity."""
if not vector_db:
return None
query_emb = vectorize_text(question)
best_match = None
best_score = 0.0
for entry in vector_db:
score = cosine(query_emb, entry["embedding"])
if score > best_score:
best_score = score
best_match = entry
if best_score >= threshold:
return best_match, best_score
else:
return None
```

```
import time  # Add at the top of your script, with other imports

if __name__ == "__main__":
questions = [
"What is the product delivery time?",
"How do I cancel my subscription?",
"Is there interest-free installment?",
"What models are available?",
"How do I update my registration data?",
"What payment methods do you accept?",
"How do I change the delivery address?",
"How do I exchange a defective product?",
"What models are available?",
"What are your support hours?",
"How do I get a second copy of my bill?",
"What are the benefits of the loyalty club?",
"How do I cancel my subscription?",  # Repeated question
"Is there interest-free installment?", # Repeated question
"What is the product delivery time?",  # Repeated question
"How do I update my registration data?", # Repeated question
"Is the website safe for purchases?",
"Do you deliver on Saturdays?",
"How do I contact support?",
"How do I get a second copy of my bill?", # Repeated question
"Do you make international deliveries?",
"What are the benefits of the loyalty club?", # Repeated question
"How do I check my balance?",
"How do I issue an electronic invoice?",
"What online courses do you offer?",
"How do I schedule a technical visit?",
"How can I change my plan?",
"I forgot my password, how do I recover it?",
"How do I make a complaint to the ombudsman?",
"What are the customer service channels?",
"How do I cancel my subscription?", # Repeated question
"How do I update my registration data?", # Repeated question
"Is there interest-free installment?", # Repeated question
"What payment methods do you accept?", # Repeated question
"How do I contact support?" # Repeated question
]
for idx, user_question in enumerate(questions, 1):
print(f"\n[{idx}] User asks: '{user_question}'")
# 1. Check if a similar question has already been answered:
match = search_similar(user_question)
if match:
print("→ Searching in the vector database (FAQ/HISTORY):")
print(f" Answer retrieved from FAQ: '{match[0]['answer']}' [Confidence: {match[1]:.2f}]")
```

```
else:
print("→ No similar answer found in FAQ. Routing to LLM model...")
# 2. Routing to the appropriate model
selected_model = router_model(user_question)
print(f"→ Selected model: {selected_model}")
# 3. Call model via OpenRouter (simulated)
answer = call_model(selected_model, user_question)
# 4. Return answer to user
print(f" Answer generated by the model: {answer}")
# 5. Store interaction in the vector database
store_interaction(user_question, answer, selected_model)
time.sleep(5)  # Wait 5 seconds before processing the next question
```

5 Prototype Operation

How the prototype works: In the example above, a user question ”What is the product delivery time?” is defined. First, **search_similar** checks if there is a similar question in the vector DB; if there were, it would return the stored answer (simulating FAQ retrieval with no model cost). If no match is found, the system routes to a model using **router_model** – in this case, a short question, so it returns ”openai/gpt-3.5-turbo”. The **call_model** function then simulates the call and produces a fake response identifying the model. This response is displayed and, then, we store the interaction in **vector_db** with its embedding. In subsequent calls, if the user repeats the same question, **search_similar** will find this log and may return the answer directly, demonstrating the efficiency of semantic FAQ/history search.

Note: In a real environment, we would replace the simulation with real calls: use HTTP libraries to send the prompt to the OpenRouter endpoint (with the defined model) and get the JSON response. Also, the similarity function would use an optimized library (e.g., Faiss, Annoy, or internal indexing of the vector database) instead of iterating in Python.

The above prototype is modular and ready to evolve; for example, we can easily replace the routing heuristic with an ML model or incorporate new data sources into the vector, without rewriting all the logic.