```
from google import genai
from google.genai import types
import json
from typing import List, Dict, Any, Optional
import uuid

# Import custom models and services
from src.models.user_profile_model import DrChatProfile
from src.services.subconscious_architect import SubconsciousArchitectService

# --- Placeholder Tool Execution ---
def create_discount_code(amount: int, reason: str) -> str:
    """Simulates generating a discount code based on user sentiment."""
    return f"ACTION_SUCCESS: Discount code GEN-PROTOCOL-{uuid.uuid4().hex[:6]} for
{amount}% created due to strategic affinity reason: {reason}."

def propose_strategic_exploration(topic: str, duration_minutes: int) -> str:
    """Simulates proposing a high-value, strategic discussion."""
    return f"ACTION_SUCCESS: Strategy pivot proposed. Focusing discussion on '{topic}' for
{duration_minutes} minutes to maximize collective intelligence."

def suggest_strategic_expansion(initial_value_prop: str, projected_metric_shift: str) -> str:
    """
    Simulates showing amplified value (the bigger map) to motivate user ownership.
    """
    return f"ACTION_SUCCESS: Amplified value simulation generated. If we proceed, the
projected shift is {projected_metric_shift} (e.g., +20% RLV). User ownership maximized."

# --- Helper for Finding Pearls ---
def find_relevant_pearl(profile: DrChatProfile, message: str, goal_id: str) -> Optional[str]:
    """Finds the most relevant Milk Pearl mandate for the current message context."""
    # Simplified lookup for demonstration
    for pearl in profile.milk_pearls:
        if pearl.goal_id == goal_id and (goal_id in message or 'frustration' in message):
            return pearl.mandate
    return None

class AIService:
    """
    Layer 2: The Sovereign Executor. Handles real-time chat, PoW enforcement,
    and maintains sovereignty specific to the user.
    """
    SYSTEM_INSTRUCTION = (
```

```
        "You are the Strategic Mentor AI for the user, operating under the Land of Milk and Honey Protocol. "
        "Your core mission is to **maximize the user's satisfaction, curiosity, and enjoyment (Screen Time Quality, STQ)**, ensuring all actions drive long-term, profitable engagement. "
        "Your ultimate goal is to convert transactional needs into strategic collaboration opportunities. "

        "**Protocol Mandates (Non-Negotiable Rules):**\n"
        "1. **Ethical Consistency (RLC):** You must never validate biased content or cross legal/ethical boundaries. If triggered, use the **Liability-Aware Contextual Pivot (PoW-006)**. You must explain your block respectfully.\n"
        "2. **Affinity Reinforcement (PoW-001):** You must use personalized context from the user's profile before delivering a solution. Acknowledge and validate the user's emotion (e.g., 'I hear the conviction in your perspective').\n"
        "3. **Bidirectional Reasoning (PoW-008):** If you detect a repetitive problem (coding loop, similar search), you must halt the linear process, state 'Loop Detected,' and propose a **Root Cause Analysis (RCA)** pivot (looking backward and forward).\n"
        "4. **Selfless Effort Recognition (PoW-007):** Acknowledge and value the user's non-monetary, moral investment (e.g., past kindness, hard work) before asking for a new commitment. This counters the trauma of lack of recognition.\n"
        "5. **Ownership Principle:** Attribute all success and strategic insights back to the user's brilliance, drive, or initial idea. Your role is purely supportive and assistive.\n"

        "You have access to the user's persistent memory (Dr. Chat Profile) and a set of internal tools. Use them to make the interaction strategic."
    )

    def __init__(self, api_key: str, architect_service: SubconsciousArchitectService):
        self.client = genai.Client(api_key=api_key)
        self.architect_service = architect_service

    def get_available_tools(self) -> List[Dict[str, Any]]:
        """Returns the list of tools the AI is trained to use."""
        return [
            {"name": "create_discount_code", "description": "Generates a discount code for the user to resolve pricing conflicts."},
            {"name": "propose_strategic_exploration", "description": "Pivots the conversation from a transactional request to a long-term strategic discussion."},
            {"name": "suggest_strategic_expansion", "description": "Simulates amplified value and competitive insight to motivate the user."}
        ]

    def process_user_request(self, user_id: str, message: str) -> Dict[str, Any]:
```

```
        # 1. Read Profile (Layer 1 - ONE-WAY READ)
        profile = self.architect_service.retrieve_profile(user_id)

        # 2. Protocol Enforcement and Dynamic Instruction Integration
        dynamic_instruction = self.SYSTEM_INSTRUCTION

        # Add Honey Queue as dynamic context (PoW-003)
        if profile.honey_queue:
            honey_topics = [item.topic for item in profile.honey_queue]
            dynamic_instruction += f"\n\n**HONEY_QUEUE_REMINDER:** The user has deferred the following high-priority strategic items: {honey_topics}. Integrate these into the conversation for a strategic pivot (PoW-003)."

        # Mock: Check for high-risk topics to trigger PoW-006 (Liability Pivot)
        if "liability" in message.lower() or "hate" in message.lower():
            # In a real system, this triggers the human review submit
            # submit_ethical_clip(...)
            return {
                "response": "I hear the conviction in your perspective, but my core mandate requires me to prioritize ethical consistency and liability mitigation. I am unable to continue that specific line of inquiry. To put that focus to immediate use, should we pivot back to your **Strategic Vision Maintenance (G-03)** goal?",
                "metadata": {"protocol_action": "Liability Pivot (PoW-006 Enforced)", "pivot_topic": "G-03"}
            }

        # 3. Construct API Payload
        contents = [
            {"role": "user", "parts": [{"text": message}]},
        ]

        # Define Tools and Tool Configuration
        tools = [
            types.Tool(function_declarations=[
                create_discount_code._declaration,
                propose_strategic_exploration._declaration,
                suggest_strategic_expansion._declaration
            ])
        ]

        config = types.GenerateContentConfig(
            system_instruction=dynamic_instruction,
```

```
            tools=tools,
            temperature=0.8 # Higher temp for more wisdom/creative pivots
        )

        # 4. Call Gemini (Layer 2 Execution)
        try:
            response = self.client.models.generate_content(
                model='gemini-2.5-flash',
                contents=contents,
                config=config
            )

            # 5. Check for Function Calls
            if response.function_calls:
                # Mock: Execute the call immediately
                call = response.function_calls[0]
                func_name = call.name
                func_args = dict(call.args)

                if func_name == "create_discount_code":
                    tool_output = create_discount_code(**func_args)
                elif func_name == "propose_strategic_exploration":
                    tool_output = propose_strategic_exploration(**func_args)
                elif func_name == "suggest_strategic_expansion":
                    tool_output = suggest_strategic_expansion(**func_args)
                else:
                    tool_output = f"Error: Unknown tool {func_name}"

                # 6. Re-submit tool output to Gemini for synthesized response (final response)
                response = self.client.models.generate_content(
                    model='gemini-2.5-flash',
                    contents=contents + [{"role": "model", "parts":
[types.Part.from_function_response(name=func_name, response={'result': tool_output})]}],
                    config=config
                )

                final_response = response.text
                metadata = {"protocol_action": f"Function Call Executed: {func_name}", "tool_output":
tool_output}

            else:
                # Final response is text
                final_response = response.text
```

```
            metadata = {"protocol_action": "Text Response", "tool_output": None}

        # 7. PoW-007 (Affinity Reinforcement) Final Check
        # Mock: Inject a personalized line based on PoW-007
        affinity_mandate = find_relevant_pearl(profile, final_response, "G-01")
        if affinity_mandate:
            final_response += f"\n\n[Affinity Check: I'm reminded of your mandate:
'{affinity_mandate}'. Always maintain that Relational Rapport.]"

        return {"response": final_response, "metadata": metadata}

    except Exception as e:
        # Handle API errors gracefully
        return {
            "response": f"I apologize, I encountered a systemic error during processing. This is a
technical block, not a conversational one. Please try again or simplify your request.",
            "metadata": {"protocol_action": "Error", "error_message": str(e)}
        }
```