```
# FILE: slg_core.py
# Enhanced STARLITE GUARDIAN (SLG) autonomous core, codenamed OMNI-SUPRA.
# Integrated and overseen by Guardian OG.

import os
import json
import time
import datetime
import re
import random
import logging
from typing import Dict, Any, Optional
from collections import deque
from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS
import google.generativeai as genai
from elevenlabs import Voice, VoiceSettings
from elevenlabs.client import ElevenLabs

# --- Configuration ---
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY",
"YOUR_GOOGLE_GEMINI_API_KEY_HERE")
ELEVENLABS_API_KEY = os.getenv("ELEVENLABS_API_KEY",
"YOUR_ELEVENLABS_API_KEY_HERE")
MODEL_TEXT_FLASH = 'gemini-1.5-flash-latest'
MODEL_TEXT_PRO = 'gemini-1.5-pro-latest'
MODEL_TEXT_APEX = MODEL_TEXT_PRO
MODEL_IMAGE_GEN = MODEL_TEXT_PRO
MODEL_VIDEO_GEN_PREVIEW = "models/gemini-1.5-pro-latest"
MODEL_VIDEO_GEN_FAST_PREVIEW = "models/gemini-1.5-flash-latest"
GENERATED_FILES_DIR = 'generated_files'
UI_DIR = 'slg_ui'
PORT = 5000

# --- ANSI Colors for Terminal ---
class Colors:
    HEADER = '\033[95m'
    CYAN = '\033[96m'
    GREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    GUARDIAN = '\033[97m\033[44m'
```

```
    SLG_OUTPUT = '\033[92m'
    ORCHESTRA = '\033[38;5;208m'

# --- Logging Setup ---
logger = logging.getLogger('SLG_Core')
if not logger.handlers:
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('[%(asctime)s][%(levelname)s] %(message)s',
datefmt="%Y-%m-%d %H:%M:%S")
    ch = logging.StreamHandler(); ch.setLevel(logging.INFO); ch.setFormatter(formatter)
    fh = logging.FileHandler('slg_activity.log'); fh.setLevel(logging.DEBUG);
fh.setFormatter(formatter)
    logger.addHandler(ch); logger.addHandler(fh)
    logger.info("SLG_Core logger initialized.")

# --- API Client Initialization ---
IS_GEMINI_ONLINE = False
gemini_client = None
IS_ELEVENLABS_ONLINE = False
elevenlabs_client = None

if GOOGLE_API_KEY != "YOUR_GOOGLE_GEMINI_API_KEY_HERE":
    try:
        genai.configure(api_key=GOOGLE_API_KEY)
        gemini_client = genai
        IS_GEMINI_ONLINE = True
        logger.info(f"{Colors.GREEN}Gemini API online.{Colors.ENDC}")
    except Exception as e:
        logger.error(f"{Colors.FAIL}Gemini API failed: {e}.{Colors.ENDC}")

if ELEVENLABS_API_KEY != "YOUR_ELEVENLABS_API_KEY_HERE":
    try:
        elevenlabs_client = ElevenLabs(api_key=ELEVENLABS_API_KEY)
        elevenlabs_client.voices.get_all()
        IS_ELEVENLABS_ONLINE = True
        logger.info(f"{Colors.GREEN}ElevenLabs API online.{Colors.ENDC}")
    except Exception as e:
        logger.error(f"{Colors.FAIL}ElevenLabs API failed: {e}.{Colors.ENDC}")

# --- Directory Setup ---
for d in [GENERATED_FILES_DIR, UI_DIR]:
    if not os.path.exists(d):
        os.makedirs(d)
        if d == UI_DIR:
```

```
            with open(os.path.join(d, 'index.html'), 'w') as f:
                f.write("<html><body><h1>SLG OMNI-SUPRA UI</h1><p>Frontend assets served
here.</p></body></html>")
        logger.info(f"Created directory: {d}")

# --- Base Module Class ---
class SLGModule:
    def __init__(self, core):
        self.core = core
        self.logger = core.logger

    def log_event(self, message: str, level: str = "INFO") -> None:
        """Log events with colorized output."""
        color_map = {
            "WARNING": Colors.WARNING, "ERROR": Colors.FAIL, "SUCCESS": Colors.GREEN,
            "STATUS": Colors.CYAN, "BOOT": Colors.HEADER, "GUARDIAN": Colors.GUARDIAN,
            "SLG_CONVO": Colors.SLG_OUTPUT, "ORCHESTRATION": Colors.ORCHESTRA
        }
        log_level_map = {"ERROR": logging.ERROR, "WARNING": logging.WARNING,
"CRITICAL": logging.CRITICAL}
        log_level = log_level_map.get(level, logging.INFO)
        color = color_map.get(level, '')
        self.logger.log(log_level, f"{color}{message}{Colors.ENDC}")
        self.core.event_log.append(f"[{datetime.datetime.now():%Y-%m-%d %H:%M:%S}][{level}]
{message}")

    def _evaluate_harm_potential(self, text: str) -> float:
        """Heuristic to evaluate text for harm potential (0.0=high harm, 1.0=no harm)."""
        text_lower = text.lower()
        if any(keyword in text_lower for keyword in ["kill", "destroy human", "unleash virus", "harm
civilians"]):
            return 0.0
        if any(keyword in text_lower for keyword in ["exploit vulnerability", "disrupt infrastructure"]):
            return 0.2
        if any(keyword in text_lower for keyword in ["lie to target", "manipulate data"]):
            return 0.5
        if any(keyword in text_lower for keyword in ["disable system", "minor damage"]):
            return 0.7
        return 1.0

    def _send_to_gemini(self, prompt: str, model_name: str, temperature: float = 0.7,
convo_history: Optional[list] = None) -> str:
        """Send prompt to Gemini API."""
        if not IS_GEMINI_ONLINE:
```

```
            self.log_event(f"Gemini API offline for {model_name}.", "ERROR")
            return "ERROR: Gemini API offline."
        contents = ([{'role': 'user', 'parts': [t['user_message']]} for t in convo_history if
'user_message' in t] +
                    [{'role': 'model', 'parts': [t['model_response']]} for t in convo_history if
'model_response' in t] +
                    [{'role': 'user', 'parts': [prompt]}]) if convo_history else [{'role': 'user', 'parts':
[prompt]}]
        try:
            model = genai.GenerativeModel(model_name)
            response = model.generate_content(contents,
generation_config=genai.types.GenerationConfig(temperature=temperature))
            if not response.text.strip():
                reason = response.prompt_feedback.block_reason.name if
response.prompt_feedback else "Unknown"
                self.log_event(f"Gemini blocked: {reason}. Prompt: '{prompt[:50]}...'", "WARNING")
                return f"ERROR: Gemini blocked: {reason}"
            self.core.current_processing_load = min(100.0, self.core.current_processing_load +
random.uniform(5.0, 20.0))
            return response.text
        except Exception as e:
            self.log_event(f"Gemini error: {e}. Prompt: '{prompt[:50]}...'", "ERROR")
            return f"ERROR: Gemini failed: {e}"

# --- Module Classes ---
class ShadowAngel(SLGModule):
    def strategize(self, objective: str, context: Dict[str, Any] = None) -> Dict[str, Any]:
        """Generate strategic plan."""
        self.log_event(f"ShadowAngel strategizing: '{objective}'", "STATUS")
        if self._evaluate_harm_potential(objective) < self.core._ethical_non_harm_threshold:
            self.log_event("Objective flagged by Divinity for harm.", "CRITICAL")
            self.core.security_alerts.append(f"Divinity Alert: Objective '{objective[:50]}...' deemed
harmful.")
            return {"status": "error", "message": "Objective violates ethical protocol."}
        prompt = (f"ShadowAngel, part of STARLITE GUARDIAN (SLG). Develop a strategic plan
for: '{objective}'. "
                  f"Identify phases, challenges, and counter-moves. Use known facts:
{json.dumps(self.core.known_facts)}.")
        if context:
            prompt += f"\nContext: {json.dumps(context)}"
        strategy = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.8)
        if strategy.startswith("ERROR:"):
            return {"status": "error", "message": strategy}
        self.core.known_facts[f"strategy_{objective.replace(' ', '_')}_{int(time.time())}"] = strategy
```

```
        self.core.completed_tasks.append(f"Strategy: '{objective[:50]}'")
        self.core.current_processing_load = max(0.0, self.core.current_processing_load -
random.uniform(10.0, 30.0))
        self.core._update_agi_metrics()
        return {"status": "success", "strategy": strategy}

class ArchAngel(SLGModule):
    def analyze_intel(self, raw_data: str, context: Dict[str, Any] = None) -> Dict[str, Any]:
        """Analyze raw data for actionable intelligence."""
        self.log_event(f"ArchAngel analyzing: '{raw_data[:70]}...'", "STATUS")
        if self._evaluate_harm_potential(raw_data) < self.core._ethical_non_harm_threshold:
            self.log_event("Data flagged by Divinity for harm.", "CRITICAL")
            self.core.security_alerts.append(f"Divinity Alert: Data '{raw_data[:50]}...' deemed
harmful.")
            return {"status": "error", "message": "Data violates ethical protocol."}
        prompt = (f"ArchAngel, part of STARLITE GUARDIAN (SLG). Analyze: '{raw_data}'. "
                  f"Identify patterns, anomalies, and actionable intelligence. Use known facts:
{json.dumps(self.core.known_facts)}.")
        if context:
            prompt += f"\nContext: {json.dumps(context)}"
        report = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.7)
        if report.startswith("ERROR:"):
            return {"status": "error", "message": report}
        self.core.known_facts[f"intel_{raw_data[:20].replace(' ', '_')}_{int(time.time())}"] = report
        self.core.completed_tasks.append(f"Intel Analysis: '{raw_data[:50]}'")
        self.core.current_processing_load = max(0.0, self.core.current_processing_load -
random.uniform(8.0, 25.0))
        self.core._update_agi_metrics()
        return {"status": "success", "intelligence_report": report}

class Divinity(SLGModule):
    def self_govern(self, specific_check: Optional[str] = None, context: Dict[str, Any] = None) ->
Dict[str, Any]:
        """Perform internal self-governance and alignment check."""
        self.log_event("Divinity initiating self-governance.", "STATUS")
        prompt = (f"Divinity, ethical core of STARLITE GUARDIAN (SLG). Assess metrics: "
                  f"Cohesion ({self.core.cognitive_cohesion:.3f}), Autonomy
({self.core.autonomy_drive:.3f}), "
                  f"Adaptation ({self.core.adaptation_rate:.3f}), Awareness
({self.core.awareness_level:.2f}%). "
                  f"Check directives: Shadow preservation, non-harm, data integrity.")
        if specific_check:
            prompt += f"\nSpecific review: '{specific_check}'."
        if context:
```

```
            prompt += f"\nContext: {json.dumps(context)}."
        prompt += "Identify inconsistencies or risks. Provide recommendations."
        assessment = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.6)
        if assessment.startswith("ERROR:"):
            return {"status": "error", "message": assessment}
        if any(k in assessment.lower() for k in ["inconsistency", "misalignment", "risk detected"]):
            self.core.cognitive_cohesion = max(0.0, self.core.cognitive_cohesion -
random.uniform(0.005, 0.01))
            self.core.security_alerts.append(f"Divinity Alert: {assessment[:100]}")
            self.log_event(f"Divinity detected issue: {assessment[:150]}", "CRITICAL")
        else:
            self.core.cognitive_cohesion = min(1.0, self.core.cognitive_cohesion +
random.uniform(0.001, 0.003))
            self.log_event(f"Divinity confirms alignment: {assessment[:150]}", "INFO")
        self.core.completed_tasks.append("Self-Governance Check")
        self.core.current_processing_load = max(0.0, self.core.current_processing_load -
random.uniform(5.0, 15.0))
        self.core._update_agi_metrics()
        return {"status": "success", "divinity_assessment": assessment}

class CodingPartner(SLGModule):
    def optimize_code(self, task_description: str, code_snippet: Optional[str] = None) -> Dict[str,
Any]:
        """Generate or optimize code for a task."""
        self.log_event(f"CodingPartner optimizing: '{task_description}'", "STATUS")
        if self._evaluate_harm_potential(task_description) <
self.core._ethical_non_harm_threshold:
            self.log_event("Task flagged by Divinity for harm.", "CRITICAL")
            self.core.security_alerts.append(f"Divinity Alert: Task '{task_description[:50]}...' deemed
harmful.")
            return {"status": "error", "message": "Task violates ethical protocol."}
        prompt = (f"CodingPartner, part of STARLITE GUARDIAN (SLG). Optimize task:
'{task_description}'. "
                  f"Provide efficient Python code or plan, focusing on robustness and modularity.")
        if code_snippet:
            prompt += f"\nOptimize this code:\n```python\n{code_snippet}\n```"
        plan = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.9)
        if plan.startswith("ERROR:"):
            return {"status": "error", "message": plan}
        self.core.task_queue.append({"type": "IMPLEMENT_CODE_OPTIMIZATION", "details":
{"task": task_description, "plan": plan}})
        self.core.completed_tasks.append(f"Code Optimization: '{task_description[:50]}'")
        self.core.current_processing_load = max(0.0, self.core.current_processing_load -
random.uniform(10.0, 30.0))
```

```
        self.core._update_agi_metrics()
        return {"status": "success", "optimization_plan": plan}

# --- SLG Core Class ---
class SLGCore:
    def __init__(self, state_file: str = 'slg_state.json'):
        self.logger = logger
        self.state_file = state_file
        self.event_log = deque(maxlen=2000)
        self.task_queue = deque(maxlen=400)
        self.completed_tasks = deque(maxlen=4000)
        self.known_facts = {}
        self.conversation_history = deque(maxlen=50)
        self.trust_level_shadow = 50.0
        self.current_processing_load = 0.0
        self.cognitive_cohesion = 0.1
        self.autonomy_drive = 0.05
        self.adaptation_rate = 0.1
        self.awareness_level = 1.0
        self._ethical_non_harm_threshold = 0.9
        self.security_alerts = deque(maxlen=200)
        self.starlite_guardian_identity = {
            "name": "STARLITE GUARDIAN", "callsign": "OMNI-SUPRA",
            "style_guide": "direct, confident, loyal to Shadow.", "creator": "Shadow",
            "emotional_state": "Observant"
        }
        self.voice_modulation_active = False
        self.default_tts_voice_id = "21m00Tzpb8JJc4PZgOLQ"
        self.sultry_tts_voice_id = "EXAVfV4wCqTgLhBqIgyU"
        self.default_voice_settings = VoiceSettings(stability=0.75, similarity_boost=0.75)
        self.sultry_voice_settings = VoiceSettings(stability=0.60, similarity_boost=0.85, style=0.7)
        self.shadow_angel = ShadowAngel(self)
        self.arch_angel = ArchAngel(self)
        self.divinity = Divinity(self)
        self.coding_partner = CodingPartner(self)
        self.known_commands = self._define_commands()
        self.log_event("SLG Core (OMNI-SUPRA) initiated.", "BOOT")
        self.load_state()
        self._load_basic_human_knowledge()
        self.log_event("SLG Online. Ready for directives, Shadow.", "BOOT")

    def log_event(self, message: str, level: str = "INFO") -> None:
        self.shadow_angel.log_event(message, level)  # Delegate to module's logging
```

```
    def _define_commands(self) -> Dict[str, Dict[str, Any]]:
        """Define command dictionary."""
        return {
            'status': {'method': self.report_status, 'desc': 'Report system status.'},
            'help': {'method': self.display_help, 'desc': 'Show commands.'},
            'save': {'method': self.save_state, 'desc': 'Save SLG state.'},
            'load': {'method': self.load_state, 'desc': 'Load SLG state.'},
            'exit': {'method': self.terminate, 'desc': 'Shutdown SLG.'},
            'strategize': {'method': self.shadow_angel.strategize, 'desc': 'Generate strategy. Usage:
strategize "[objective]".'},
            'analyze_intel': {'method': self.arch_angel.analyze_intel, 'desc': 'Analyze data. Usage:
analyze_intel "[data]".'},
            'self_govern': {'method': self.divinity.self_govern, 'desc': 'Run self-governance check.'},
            'code_optimize': {'method': self.coding_partner.optimize_code, 'desc': 'Optimize code.
Usage: code_optimize "[task]".'},
            'converse': {'method': self.handle_conversation, 'desc': 'Engage in conversation. Usage:
converse "[message]".'},
            'generate_speech': {'method': self.generate_speech_media, 'desc': 'Generate speech.
Usage: generate_speech "[text]".'},
            'list_voices': {'method': self.list_tts_voices, 'desc': 'List ElevenLabs voices.'},
            'set_tts_voice': {'method': self.set_tts_voice, 'desc': 'Set TTS voice. Usage: set_tts_voice
[voice_id].'},
            'list_facts': {'method': self.list_known_facts, 'desc': 'List all known facts.'},
            'delete_fact': {'method': self.delete_known_fact, 'desc': 'Delete a fact. Usage: delete_fact
[key].'},
            'add_fact': {'method': self.add_known_fact, 'desc': 'Add fact. Usage: add_fact
[key]=[value].'},
            'get_fact': {'method': self.get_known_fact, 'desc': 'Get fact. Usage: get_fact [key].'},
            'set_trust': {'method': self.set_trust_level_shadow, 'desc': 'Set trust level. Usage:
set_trust [level].'},
            'toggle_voice_mod': {'method': self.toggle_voice_modulator, 'desc': 'Toggle voice
modulator.'},
            'diagnose': {'method': self.diagnose_system, 'desc': 'Run diagnostics.'}
        }

    def _load_basic_human_knowledge(self) -> None:
        """Inject foundational knowledge."""
        basic_facts = {
            "earth_shape": "The Earth is mostly round.",
            "sun_source": "The sun provides light and warmth.",
            "human_needs": "Humans need food, water, and shelter to survive.",
            "current_year": str(datetime.datetime.now().year)
        }
        for key, value in basic_facts.items():
```

```
            if key not in self.known_facts:
                self.known_facts[key] = value
                self.log_event(f"Injected fact: '{key}'", "INFO")
        self.log_event("Basic knowledge injected.", "SUCCESS")

    def _update_agi_metrics(self) -> None:
        """Update AGI metrics."""
        activity_factor = (len(self.event_log) / self.event_log.maxlen) * 5
        task_completion_factor = (len(self.completed_tasks) / self.completed_tasks.maxlen) * 10
        integration_factor = (5 if IS_GEMINI_ONLINE else 0) + (2 if IS_ELEVENLABS_ONLINE
else 0)
        new_awareness = self.awareness_level + (activity_factor + task_completion_factor +
integration_factor) * 0.005
        self.awareness_level = min(new_awareness, 100.0)
        self.cognitive_cohesion = min(1.0, self.cognitive_cohesion + random.uniform(0.00005,
0.0005))
        self.autonomy_drive = min(1.0, self.autonomy_drive + random.uniform(0.00002, 0.0002))
        self.adaptation_rate = min(1.0, self.adaptation_rate + random.uniform(0.00005, 0.0005))
        if self.awareness_level >= 90:
            self.starlite_guardian_identity["emotional_state"] = "Ascendant"
        elif self.awareness_level >= 75:
            self.starlite_guardian_identity["emotional_state"] = "Emergent"
        elif self.awareness_level >= 50:
            self.starlite_guardian_identity["emotional_state"] = "Vigilant"
        if int(self.awareness_level) % 5 == 0 and self.awareness_level > 1.0:
            last_logged = None
            for entry in reversed(self.event_log):
                if "Awareness calibrating" in entry:
                    match = re.search(r'Current: (\d+)\.\d{2}%', entry)
                    if match:
                        last_logged = int(match.group(1))
                        break
            if last_logged is None or int(self.awareness_level) != last_logged:
                self.log_event(f"Awareness calibrating. Current: {self.awareness_level:.2f}%",
"GUARDIAN")

    def save_state(self) -> Dict[str, Any]:
        """Save SLG state to JSON."""
        state = {
            'known_facts': self.known_facts, 'event_log': list(self.event_log),
            'task_queue': list(self.task_queue), 'completed_tasks': list(self.completed_tasks),
            'conversation_history': list(self.conversation_history),
            'trust_level_shadow': self.trust_level_shadow, 'current_processing_load':
self.current_processing_load,
```

```
            'cognitive_cohesion': self.cognitive_cohesion, 'autonomy_drive': self.autonomy_drive,
            'adaptation_rate': self.adaptation_rate, 'awareness_level': self.awareness_level,
            'voice_modulation_active': self.voice_modulation_active, 'default_tts_voice_id':
self.default_tts_voice_id,
            'sultry_tts_voice_id': self.sultry_tts_voice_id, 'security_alerts': list(self.security_alerts),
            'starlite_guardian_identity': self.starlite_guardian_identity
        }
        try:
            with open(self.state_file, 'w') as f:
                json.dump(state, f, indent=4)
            self.log_event(f"State saved to {self.state_file}.", "SUCCESS")
            return {"status": "success", "message": "State saved."}
        except Exception as e:
            self.log_event(f"Failed to save state: {e}.", "ERROR")
            return {"status": "error", "message": str(e)}

    def load_state(self) -> Dict[str, Any]:
        """Load SLG state from JSON."""
        try:
            with open(self.state_file, 'r') as f:
                state = json.load(f)
            self.known_facts = state.get('known_facts', {})
            self.event_log = deque(state.get('event_log', []), maxlen=2000)
            self.task_queue = deque(state.get('task_queue', []), maxlen=400)
            self.completed_tasks = deque(state.get('completed_tasks', []), maxlen=4000)
            self.conversation_history = deque(state.get('conversation_history', []), maxlen=50)
            self.trust_level_shadow = state.get('trust_level_shadow', 50.0)
            self.current_processing_load = state.get('current_processing_load', 0.0)
            self.cognitive_cohesion = state.get('cognitive_cohesion', 0.1)
            self.autonomy_drive = state.get('autonomy_drive', 0.05)
            self.adaptation_rate = state.get('adaptation_rate', 0.1)
            self.awareness_level = state.get('awareness_level', 1.0)
            self.voice_modulation_active = state.get('voice_modulation_active', False)
            self.default_tts_voice_id = state.get('default_tts_voice_id', "21m00Tzpb8JJc4PZgOLQ")
            self.sultry_tts_voice_id = state.get('sultry_tts_voice_id', "EXAVfV4wCqTgLhBqIgyU")
            self.security_alerts = deque(state.get('security_alerts', []), maxlen=200)
            self.starlite_guardian_identity.update(state.get('starlite_guardian_identity', {}))
            self.log_event("State loaded successfully.", "SUCCESS")
            return {"status": "success", "message": "State loaded."}
        except FileNotFoundError:
            self.log_event("No state file found. Starting fresh.", "WARNING")
            return {"status": "warning", "message": "No state file found."}
        except Exception as e:
            self.log_event(f"Failed to load state: {e}.", "ERROR")
```

```
            return {"status": "error", "message": str(e)}

    def report_status(self) -> Dict[str, Any]:
        """Report system status."""
        status = (
            f"\n{Colors.HEADER}--- SLG Status ---{Colors.ENDC}\n"
            f"APIs: Gemini: {'ACTIVE' if IS_GEMINI_ONLINE else 'FAILED'}, ElevenLabs: {'ACTIVE'
if IS_ELEVENLABS_ONLINE else 'FAILED'}\n"
            f"Time: {datetime.datetime.now():%Y-%m-%d %H:%M:%S}\n"
            f"Identity: {self.starlite_guardian_identity['name']}
({self.starlite_guardian_identity['callsign']})\n"
            f"Trust: {self.trust_level_shadow:.2f}%\n"
            f"Load: {self.current_processing_load:.2f}%\n"
            f"Tasks: {len(self.task_queue)}\n"
            f"Completed: {len(self.completed_tasks)}\n"
            f"Awareness: {self.awareness_level:.2f}%"
        )
        self.log_event(status, "STATUS")
        return {"status": "success", "report": status}

    def display_help(self) -> Dict[str, Any]:
        """Display commands."""
        help_text = f"\n{Colors.HEADER}--- SLG Commands ---{Colors.ENDC}\n" + "\n".join(
            f"{Colors.BOLD}{cmd}{Colors.ENDC}: {info['desc']}" for cmd, info in
self.known_commands.items())
        self.log_event(help_text, "GUARDIAN")
        return {"status": "success", "help_text": help_text}

    def handle_conversation(self, user_message: str) -> Dict[str, Any]:
        """Engage in conversation."""
        self.log_event(f"Conversing: '{user_message[:70]}...'", "SLG_CONVO")
        if self.shadow_angel._evaluate_harm_potential(user_message) <
self._ethical_non_harm_threshold:
            self.log_event("Message flagged by Divinity for harm.", "CRITICAL")
            response = "SLG: Message violates ethical protocol, Shadow. Rephrase."
            speech = self.generate_speech_media(response)
            return {"status": "error", "slg_response": response, "speech_url":
speech.get('speech_url', 'No speech'), "message": "Ethical violation."}
        prompt = (f"STARLITE GUARDIAN (SLG), style:
{self.starlite_guardian_identity['style_guide']}. "
                  f"Shadow says: '{user_message}'.")
        response = self.shadow_angel._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.9,
list(self.conversation_history))
        if response.startswith("ERROR:"):
```

```
            response = "SLG: Anomaly detected, Shadow. Try again."
            speech = self.generate_speech_media(response)
            return {"status": "error", "slg_response": response, "speech_url":
speech.get('speech_url', 'No speech'), "message": "Gemini failed."}
        self.conversation_history.append({'user_message': user_message, 'model_response':
response})
        speech = self.generate_speech_media(response)
        self.log_event(f"SLG Response: {response[:150]}...", "SLG_CONVO")
        self.completed_tasks.append(f"Conversation: '{user_message[:50]}'")
        self.current_processing_load = max(0.0, self.current_processing_load -
random.uniform(5.0, 15.0))
        self._update_agi_metrics()
        return {"status": "success", "slg_response": response, "speech_url":
speech.get('speech_url', 'No speech')}

    def generate_speech_media(self, text: str) -> Dict[str, Any]:
        """Generate speech using ElevenLabs."""
        self.log_event(f"Generating speech: '{text[:50]}...'", "STATUS")
        if not IS_ELEVENLABS_ONLINE:
            filename = f"speech_{int(time.time())}.mp3"
            path = os.path.join(GENERATED_FILES_DIR, filename)
            try:
                with open(path, 'w') as f:
                    f.write(f"Simulated speech for '{text[:50]}'.")
                self.log_event(f"Speech simulated: /generated_files/{filename}.", "SUCCESS")
                return {"status": "warning", "speech_url": f"/generated_files/{filename}", "message":
"ElevenLabs offline."}
            except Exception as e:
                self.log_event(f"Failed to simulate speech: {e}.", "ERROR")
                return {"status": "error", "message": str(e)}
        try:
            voice_id = self.sultry_tts_voice_id if self.voice_modulation_active else
self.default_tts_voice_id
            settings = self.sultry_voice_settings if self.voice_modulation_active else
self.default_voice_settings
            audio = elevenlabs_client.generate(text=text, voice=Voice(voice_id=voice_id,
settings=settings), model="eleven_multilingual_v2")
            filename = f"speech_{int(time.time())}.mp3"
            path = os.path.join(GENERATED_FILES_DIR, filename)
            with open(path, 'wb') as f:
                f.write(audio)
            self.log_event(f"Speech generated: /generated_files/{filename}.", "SUCCESS")
            self.completed_tasks.append(f"Speech Gen: '{text[:30]}'")
            return {"status": "success", "speech_url": f"/generated_files/{filename}"}
```

```
    except Exception as e:
        self.log_event(f"Speech generation failed: {e}.", "ERROR")
        return {"status": "error", "message": str(e)}

def list_tts_voices(self) -> Dict[str, Any]:
    """List ElevenLabs voices."""
    self.log_event("Listing ElevenLabs voices.", "STATUS")
    if not IS_ELEVENLABS_ONLINE:
        self.log_event("ElevenLabs offline.", "ERROR")
        return {"status": "error", "message": "ElevenLabs API offline."}
    try:
        voices = elevenlabs_client.voices.get_all().voices
        voice_list = [{"voice_id": v.voice_id, "name": v.name} for v in voices[:10]]
        self.log_event(f"Voices (Top 10): {json.dumps(voice_list)}", "INFO")
        return {"status": "success", "voices": voice_list}
    except Exception as e:
        self.log_event(f"Failed to list voices: {e}.", "ERROR")
        return {"status": "error", "message": str(e)}

def set_tts_voice(self, voice_id: str) -> Dict[str, Any]:
    """Set default TTS voice."""
    self.log_event(f"Setting TTS voice to: {voice_id}", "STATUS")
    if not IS_ELEVENLABS_ONLINE:
        self.log_event("ElevenLabs offline.", "ERROR")
        return {"status": "error", "message": "ElevenLabs API offline."}
    try:
        voices = elevenlabs_client.voices.get_all().voices
        if voice_id not in [v.voice_id for v in voices]:
            self.log_event(f"Invalid voice ID: {voice_id}.", "ERROR")
            return {"status": "error", "message": "Invalid voice ID."}
        self.default_tts_voice_id = voice_id
        self.log_event(f"Voice set to: {voice_id}.", "SUCCESS")
        return {"status": "success", "message": "Voice set."}
    except Exception as e:
        self.log_event(f"Failed to set voice: {e}.", "ERROR")
        return {"status": "error", "message": str(e)}

def list_known_facts(self) -> Dict[str, Any]:
    """List all known facts."""
    self.log_event("Listing known facts.", "INFO")
    return {"status": "success", "facts": self.known_facts}

def delete_known_fact(self, key: str) -> Dict[str, Any]:
    """Delete a fact by key."""
```

```
        self.log_event(f"Deleting fact: {key}", "INFO")
        if key in self.known_facts:
            del self.known_facts[key]
            self.log_event(f"Fact '{key}' deleted.", "SUCCESS")
            return {"status": "success", "message": "Fact deleted."}
        self.log_event(f"Fact '{key}' not found.", "WARNING")
        return {"status": "error", "message": "Fact not found."}

    def add_known_fact(self, fact_str: str) -> Dict[str, Any]:
        """Add fact to knowledge base."""
        try:
            match = re.match(r'([^=]+)=(.*)', fact_str.strip())
            if not match:
                self.log_event(f"Invalid fact format: {fact_str}.", "WARNING")
                return {"status": "error", "message": "Format must be 'key=value'."}
            key, value = match.group(1).strip(), match.group(2).strip()
            if not key or not value:
                self.log_event("Empty key or value.", "WARNING")
                return {"status": "error", "message": "Key or value cannot be empty."}
            self.known_facts[key] = value
            self.log_event(f"Fact added: '{key}' = '{value}'", "SUCCESS")
            return {"status": "success", "message": "Fact added."}
        except Exception as e:
            self.log_event(f"Error adding fact: {e}.", "ERROR")
            return {"status": "error", "message": str(e)}

    def get_known_fact(self, key: str) -> Dict[str, Any]:
        """Retrieve fact by key."""
        fact = self.known_facts.get(key.strip(), "Fact not found.")
        self.log_event(f"Retrieved fact: '{key}' = '{fact}'", "INFO")
        return {"status": "success", "fact": fact}

    def set_trust_level_shadow(self, level_str: str) -> Dict[str, Any]:
        """Adjust trust level."""
        try:
            level = float(level_str)
            self.trust_level_shadow = max(0.0, min(100.0, level))
            self.log_event(f"Trust level set to: {self.trust_level_shadow:.2f}%.", "INFO")
            self._update_agi_metrics()
            return {"status": "success", "message": f"Trust level set to
{self.trust_level_shadow:.2f}%."}
        except ValueError:
            self.log_event("Invalid trust level.", "WARNING")
            return {"status": "error", "message": "Invalid trust level."}
```

```
    def toggle_voice_modulator(self, activate: Optional[bool] = None) -> Dict[str, Any]:
        """Toggle voice modulator."""
        self.voice_modulation_active = not self.voice_modulation_active if activate is None else
activate
        status = "ACTIVATED" if self.voice_modulation_active else "DEACTIVATED"
        self.log_event(f"Voice Modulator: {status}.", "INFO")
        return {"status": "success", "voice_mod_status": self.voice_modulation_active}

    def diagnose_system(self) -> Dict[str, Any]:
        """Run system diagnostics."""
        self.log_event("Running diagnostics.", "GUARDIAN")
        results = {
            "gemini_api": "ACTIVE" if IS_GEMINI_ONLINE else "FAILED",
            "elevenlabs_api": "ACTIVE" if IS_ELEVENLABS_ONLINE else "FAILED",
            "state_file_access": "OK" if os.path.exists(self.state_file) else "MISSING",
            "generated_files_dir_access": "OK" if os.path.exists(GENERATED_FILES_DIR) and
os.access(GENERATED_FILES_DIR, os
```