Synthetic Reconbot Mechanism Using Adaptive Learning

Infosight AI:

Code:

```
from flask import Blueprint, request, jsonify, render_template, send_file
from flask_cors import CORS
import requests
import base64
import google.generativeai as genai
import logging
from PIL import Image
import io
import re
from datetime import datetime, timedelta
from collections import deque
import threading
import time

# Create blueprint
infosight_ai = Blueprint('infosight_ai', __name__, template_folder='templates')
logger = logging.getLogger(__name__)
CORS(infosight_ai)

# API Configuration
GEMINI_API_KEY = "AIzaSyCHIGVlOgUmiPy-_1UFjqlXUWkfrvDWc1k"
HF_API_TOKEN = "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"

class RateLimiter:
    def __init__(self, max_requests, time_window):
        self.max_requests = max_requests
```

```
        self.time_window = time_window
        self.requests = deque()
        self.lock = threading.Lock()

    def can_proceed(self):
        now = datetime.now()
        with self.lock:
            while self.requests and self.requests[0] < now -
timedelta(seconds=self.time_window):
                self.requests.popleft()

            if len(self.requests) < self.max_requests:
                self.requests.append(now)
                return True
            return False

    def wait_time(self):
        if not self.requests:
            return 0
        now = datetime.now()
        oldest_request = self.requests[0]
        return max(0, (oldest_request + timedelta(seconds=self.time_window) -
now).total_seconds())

class AIGenerator:
    def __init__(self):
        genai.configure(api_key=GEMINI_API_KEY)
        self.gemini_model = genai.GenerativeModel('gemini-pro')
        self.hf_model = "CompVis/stable-diffusion-v1-4"  # Changed to more reliable free
model
```

```
        self.rate_limiter = RateLimiter(max_requests=10, time_window=60)

    def format_text_content(self, text: str) -> str:
        text = re.sub(r'\*+', '', text)
        sections = []
        current_section = []

        for line in text.strip().split('\n'):
            line = line.strip()
            if not line:
                if current_section:
                    sections.append('\n'.join(current_section))
                    current_section = []
            else:
                line = re.sub(r'[#_~]', '', line)
                current_section.append(line)

        if current_section:
            sections.append('\n'.join(current_section))

        return '\n\n'.join(sections)

    def generate_text(self, prompt: str) -> str:
        try:
            enhanced_prompt = f"""
            Provide information about {prompt}. Include:
            1. A clear introduction
            2. Key characteristics and features
            3. Interesting and unique aspects
```

```
        4. Practical applications or relevant details

        Please provide the information in clear paragraphs without using any special characters, asterisks, or markdown formatting.
        Keep the tone professional and informative.
        """

        response = self.gemini_model.generate_content(enhanced_prompt)
        if not response.text:
            raise ValueError("No text generated from the model")

        return self.format_text_content(response.text)

    except Exception as e:
        logger.error(f"Text generation error: {str(e)}")
        raise

def generate_image(self, prompt: str, retry_count=0, max_retries=3) -> bytes:
    try:
        if not self.rate_limiter.can_proceed():
            wait_time = self.rate_limiter.wait_time()
            raise ValueError(f"Rate limit exceeded. Please wait {wait_time:.1f} seconds.")

        enhanced_prompt = (
            f"A highly detailed visualization of {prompt}, "
            "professional quality, sharp focus, perfect lighting"
        )

        api_url = f"https://api-inference.huggingface.co/models/{self.hf_model}"
        headers = {"Authorization": f"Bearer {HF_API_TOKEN}"}
```

```
# Optimized parameters for faster generation
payload = {
    "inputs": enhanced_prompt,
    "parameters": {
        "num_inference_steps": 20,  # Reduced for faster generation
        "guidance_scale": 7.0,
        "width": 512,  # Standard size for faster generation
        "height": 512
    }
}

# Check model status first
status_response = requests.get(api_url, headers=headers, timeout=10)
if "error" in status_response.json():
    if retry_count < max_retries:
        time.sleep(5)  # Wait 5 seconds before retrying
        return self.generate_image(prompt, retry_count + 1, max_retries)
    else:
        raise ValueError("Model is currently unavailable. Please try again later.")

response = requests.post(
    api_url,
    headers=headers,
    json=payload,
    timeout=30
)

if response.status_code != 200:
```

```
            error_msg = response.text if response.text else "Unknown error"
            if retry_count < max_retries:
                time.sleep(5)
                return self.generate_image(prompt, retry_count + 1, max_retries)
            else:
                raise ValueError(f"Image generation failed after {max_retries} attempts: {error_msg}")

        return response.content

    except Exception as e:
        logger.error(f"Image generation error: {str(e)}")
        if retry_count < max_retries:
            time.sleep(5)
            return self.generate_image(prompt, retry_count + 1, max_retries)
        raise

def generate_both(self, prompt: str):
    try:
        if not self.rate_limiter.can_proceed():
            wait_time = self.rate_limiter.wait_time()
            raise ValueError(f"Rate limit exceeded. Please wait {wait_time:.1f} seconds.")

        # Generate text first as it's more reliable
        text = self.generate_text(prompt)

        # Then try image generation with retries
        image = self.generate_image(prompt)

        return text, image
```

```
        except Exception as e:
            logger.error(f"Combined generation error: {str(e)}")
            raise

# Initialize generator
generator = AIGenerator()

@infosight_ai.route('/')
def index():
    return render_template('infosight_ai.html')

@infosight_ai.route('/generate-text', methods=['POST'])
def generate_text():
    try:
        data = request.get_json()
        if not data or 'prompt' not in data:
            return jsonify({'error': 'No prompt provided'}), 400

        text = generator.generate_text(data['prompt'])
        return jsonify({'text': text})
    except Exception as e:
        logger.error(f"Text generation endpoint error: {str(e)}")
        return jsonify({'error': str(e)}), 500

@infosight_ai.route('/generate-image', methods=['POST'])
def generate_image():
    try:
        data = request.get_json()
        if not data or 'prompt' not in data:
```

```
            return jsonify({'error': 'No prompt provided'}), 400

        image_bytes = generator.generate_image(data['prompt'])
        if not image_bytes:
            return jsonify({'error': 'Image generation failed'}), 500

        image_base64 = base64.b64encode(image_bytes).decode('utf-8')
        return jsonify({'image_url': f"{image_base64}"})
    except ValueError as ve:
        return jsonify({'error': str(ve)}), 429
    except Exception as e:
        logger.error(f"Image generation endpoint error: {str(e)}")
        return jsonify({'error': str(e)}), 500

@infosight_ai.route('/generate-both', methods=['POST'])
def generate_both():
    try:
        data = request.get_json()
        if not data or 'prompt' not in data:
            return jsonify({'error': 'No prompt provided'}), 400

        text, image_bytes = generator.generate_both(data['prompt'])
        response = {'text': text}

        if image_bytes:
            image_base64 = base64.b64encode(image_bytes).decode('utf-8')
            response['image_url'] = f"{image_base64}"

        return jsonify(response)
```

```
    except ValueError as ve:
        return jsonify({'error': str(ve)}), 429
    except Exception as e:
        logger.error(f"Combined generation endpoint error: {str(e)}")
        return jsonify({'error': str(e)}), 500
```

Explanation:

I'll provide a well-structured explanation of this Flask backend code that integrates with Gemini and Hugging Face APIs for text and image generation.

1) Packages - Usage

```python
from flask import Blueprint, request, jsonify, render_template, send_file
from flask_cors import CORS
import requests
import base64
import google.generativeai as genai
import logging
from PIL import Image
import io
import re
from datetime import datetime, timedelta
from collections import deque
import threading
import time
```

- **Flask Components**: Used to create routes (`Blueprint`), handle requests, return JSON responses, and render templates

- **CORS**: Enables cross-origin resource sharing, allowing frontend from different domains to interact with this API

- **requests**: For making HTTP requests to external APIs (Hugging Face)

- **base64**: For encoding image data to be sent as text in JSON responses

- **google.generativeai**: Google's Gemini API client library

- **logging**: For error tracking and debugging

- **PIL (Pillow)**: For image processing (though not heavily used in this code)

- **re**: For regex pattern matching to clean text

- **datetime/timedelta**: For timestamp handling in rate limiting

- **deque**: Efficient collection for tracking request timestamps

- **threading**: For thread-safe operations in rate limiting

- **time**: For implementing delays between retries

2) Classes and Functions - Usage and Explanation

Blueprint Setup

```python
infosight_ai = Blueprint('infosight_ai', __name__, template_folder='templates')
logger = logging.getLogger(__name__)
CORS(infosight_ai)
```

- Creates a Flask Blueprint named 'infosight_ai' with its own template folder

- Sets up logging and enables CORS for this blueprint

API Configuration

```python
GEMINI_API_KEY = "AIzaSyCHIGVlOgUmiPy-_1UFjqlXUWkfrvDWc1k"
HF_API_TOKEN = "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"
```

- API keys for Google's Gemini and Hugging Face services

- **Security Note**: Hardcoding API keys in source code is not recommended; they should be stored in environment variables

`RateLimiter` Class

```python
class RateLimiter:
    def __init__(self, max_requests, time_window):
        self.max_requests = max_requests
        self.time_window = time_window
        self.requests = deque()
        self.lock = threading.Lock()
```

- Implements a simple rate limiting mechanism
- `max_requests`: Maximum number of allowed requests within the time window
- `time_window`: Time period (in seconds) for rate limiting
- `requests`: Queue of timestamps for tracking recent requests
- `lock`: Thread lock for thread-safe operations

```python
def can_proceed(self):
    # Implementation details...
```

- Checks if a new request can proceed based on rate limits
- Removes expired timestamps outside the time window
- Returns `True` if request can proceed, `False` otherwise

```python
def wait_time(self):
    # Implementation details...
```

- Calculates remaining wait time until next request can be made
- Used to provide meaningful feedback to users when rate limited

`AIGenerator` Class

```python
class AIGenerator:
    def __init__(self):
        genai.configure(api_key=GEMINI_API_KEY)
        self.gemini_model = genai.GenerativeModel('gemini-pro')
        self.hf_model = "CompVis/stable-diffusion-v1-4"
        self.rate_limiter = RateLimiter(max_requests=10, time_window=60)
```

- Main class that handles AI generation operations
- Configures Gemini API with the provided key
- Initializes Gemini Pro model for text generation
- Specifies Stable Diffusion v1.4 model for image generation
- Creates rate limiter allowing 10 requests per minute

```python
def format_text_content(self, text: str) -> str:
    # Implementation details...
```

- Cleans and formats text generated by Gemini
- Removes special characters, asterisks, and markdown formatting
- Organizes text into proper paragraphs with consistent spacing

```python
def generate_text(self, prompt: str) -> str:
    # Implementation details...
```

```

- Generates informative text about a given prompt using Gemini
- Enhances the original prompt with structured guidelines
- Formats the response and handles potential errors

```python
def generate_image(self, prompt: str, retry_count=0, max_retries=3) -> bytes:
    # Implementation details...
```

- Generates an image for a given prompt using Hugging Face's Stable Diffusion API
- Implements retry logic with up to 3 attempts
- Checks model availability before full generation request
- Uses optimized parameters for faster generation
- Returns raw image bytes on success

```python
def generate_both(self, prompt: str):
    # Implementation details...
```

- Convenience method that generates both text and image for the same prompt
- Prioritizes text generation first (as it's more reliable)
- Returns both results together

3) Routes - Usage and Explanation

```python
@infosight_ai.route('/')
def index():
    return render_template('infosight_ai.html')
```
```

```

- Root route that renders the main infosight_ai.html template
- Serves as the landing page for the application

```python
@infosight_ai.route('/generate-text', methods=['POST'])
def generate_text():
    # Implementation details...
```

- Endpoint for text generation only
- Accepts POST requests with JSON body containing a 'prompt' field
- Returns generated text in JSON format
- Includes error handling for various failure scenarios

```python
@infosight_ai.route('/generate-image', methods=['POST'])
def generate_image():
    # Implementation details...
```

- Endpoint for image generation only
- Accepts POST requests with JSON body containing a 'prompt' field
- Returns base64-encoded image data as a data URL in JSON format
- Includes special handling for rate limit errors (429 status)

```python
@infosight_ai.route('/generate-both', methods=['POST'])
def generate_both():
    # Implementation details...
```
```

- Endpoint that generates both text and image for the same prompt
- Returns a combined JSON response with both text and image data
- Uses the same error handling patterns as the individual endpoints

4) Other Details - Clear Explanation

Error Handling Strategy
- The code implements comprehensive error handling at multiple levels:
  - General exception handling in route handlers
  - Specific exception handling in generator methods
  - Custom error responses with appropriate HTTP status codes
  - Detailed logging for debugging purposes

Rate Limiting Implementation
- Prevents abuse and API quota exhaustion
- Thread-safe implementation using locks
- Provides informative feedback about wait times when limits are hit

Retry Mechanism
- Especially important for image generation which may occasionally fail
- Implements exponential backoff (fixed 5-second delay between retries)
- Limited to a maximum of 3 attempts to prevent infinite retries

Prompt Enhancement
- Both text and image generation use "enhanced prompts" that expand on user input
- For text: Structures the response with introduction, features, applications, etc.
- For images: Adds qualifiers like "highly detailed," "professional quality," etc.

Image Optimization

- Uses reduced parameters for Stable Diffusion to improve generation speed:

  - Fewer inference steps (20 instead of the default 50)

  - Standard dimensions (512×512)

5) Recap of the Code

This Flask blueprint implements an AI generation service with the following key features:

1. **Dual AI Integration**: Leverages Google's Gemini API for text generation and Hugging Face's Stable Diffusion for image generation

2. **Rate Limiting**: Implements a thread-safe rate limiter to prevent API abuse and quota exhaustion

3. **Robust Error Handling**: Provides appropriate error responses and implements retry logic for unreliable services

4. **Prompt Enhancement**: Automatically improves user prompts to generate better results

5. **Combined Generation**: Offers both individual and combined text/image generation endpoints

6. **Performance Optimization**: Uses parameter tuning for faster image generation

The code follows good practices for a production API including:

- Comprehensive error handling and logging

- Rate limiting to prevent abuse

- Clean response formatting

- CORS support for cross-origin requests

However, it could be improved by:

- Moving API keys to environment variables

- Implementing authentication

- Adding input validation and sanitization

- Considering asynchronous processing for long-running image generation

This implementation provides a solid foundation for an AI-powered content generation service that could be integrated into various applications.

2) LANA_AI:
CODE:

```
import os
from flask import Flask, render_template, jsonify, Blueprint
import speech_recognition as sr
import pygame
from gtts import gTTS
from time import time
import threading
import signal
import numpy as np
from typing import Optional, Tuple

# Create a blueprint
lana_ai = Blueprint('lana_ai', __name__, template_folder='templates')

# API key
GOOGLE_API_KEY = 'AIzaSyBfsdtO0ipw31w1kLbuUn5_zpHZCJeLXI8'

# Initialize APIs
from google.generativeai import configure, GenerativeModel

configure(api_key=GOOGLE_API_KEY)
model = GenerativeModel('gemini-pro')
pygame.mixer.init()
```

```
# Define constants
RECORDING_PATH = "audio/recording.wav"
RESPONSE_PATH = "audio/response.mp3"
PROMPT_TEMPLATE = "You are Lana, Boss human assistant. You are witty and full of personality. Your answers should be limited to 3 lines short sentences.\nBoss: {user_input}\nLana: "

# Initialize Flask app
app = Flask(__name__)
is_listening = False
thread = None
latest_transcription = ""
latest_response = ""
conversation_lock = threading.Lock()
stop_event = threading.Event()

# Audio visualization data
audio_data = np.array([])

def log(message: str):
    """Print and write to status.txt"""
    print(message)
    with open("status.txt", "a") as f:
        f.write(f"{time()}: {message}\n")

def request_gemini(prompt: str) -> str:
    """Generate content using the Gemini model"""
    try:
        response = model.generate_content(prompt)
```

```
        return response.text
    except Exception as e:
        log(f"Error generating Gemini response: {e}")
        return "I'm having trouble thinking right now. Could you repeat that?"

def enhanced_record_audio(timeout: int = 5, phrase_timeout: int = 10) ->
Tuple[Optional[str], np.ndarray]:
    """
    Enhanced audio recording with better error handling and audio level monitoring.
    """
    recognizer = sr.Recognizer()
    audio_data = np.array([])

    # Customize speech recognition parameters
    recognizer.energy_threshold = 300
    recognizer.dynamic_energy_threshold = True
    recognizer.pause_threshold = 0.8

    try:
        with sr.Microphone() as source:
            log("Adjusting for ambient noise...")
            recognizer.adjust_for_ambient_noise(source, duration=2)

            log(f"Energy threshold set to {recognizer.energy_threshold}")
            log("Listening...")

            try:
                audio = recognizer.listen(source,
                                          timeout=timeout,
                                          phrase_time_limit=phrase_timeout)
```

```
        # Convert audio to numpy array for visualization
        audio_data = np.frombuffer(audio.get_raw_data(), dtype=np.int16)

        # Save the audio file
        with open(RECORDING_PATH, "wb") as f:
            f.write(audio.get_wav_data())

        try:
            # Try multiple recognition attempts
            text = None

            try:
                text = recognizer.recognize_google(audio)
            except sr.UnknownValueError:
                log("First attempt failed, trying with more aggressive noise reduction...")
                recognizer.energy_threshold = 400
                text = recognizer.recognize_google(audio)

            if text:
                log(f"Transcribed text: {text}")
                return text, audio_data

        except sr.RequestError as e:
            log(f"Google Speech Recognition service error: {e}")
            return None, audio_data

    except sr.WaitTimeoutError:
        log("Listening timed out. No speech detected.")
```

```
            return None, audio_data

    except Exception as e:
        log(f"Recording error: {e}")
        return None, audio_data

    return None, audio_data

def record_audio() -> str:
    """Main recording function"""
    text, new_audio_data = enhanced_record_audio()
    global audio_data
    audio_data = new_audio_data
    return "Recording complete" if text else "No speech detected"

def transcribe_audio() -> str:
    """Transcribe saved audio file using Google's speech recognition"""
    recognizer = sr.Recognizer()
    try:
        with sr.AudioFile(RECORDING_PATH) as source:
            audio = recognizer.record(source)
            return recognizer.recognize_google(audio)
    except sr.UnknownValueError:
        log("Google Speech Recognition could not understand audio")
        return ""
    except sr.RequestError as e:
        log(f"Google Speech Recognition service error: {e}")
        return ""
    except Exception as e:
```

```
        log(f"Transcription error: {e}")
        return ""

def listen_and_respond():
    """Main loop for listening and responding"""
    global latest_transcription, latest_response
    while not stop_event.is_set():
        try:
            # Record audio
            record_status = record_audio()
            if record_status != "Recording complete":
                continue

            # Check if stop event is set
            if stop_event.is_set():
                break

            # Transcribe audio
            words = transcribe_audio()
            if not words:
                continue

            # Update latest transcription
            with conversation_lock:
                latest_transcription = words

            # Get response from Gemini
            prompt = PROMPT_TEMPLATE.format(user_input=words)
            response = request_gemini(prompt)
```

```
            # Update latest response
            with conversation_lock:
                latest_response = response

            # Convert response to audio and play it
            try:
                tts = gTTS(response)
                tts.save(RESPONSE_PATH)
                sound = pygame.mixer.Sound(RESPONSE_PATH)
                sound.play()
                pygame.time.wait(int(sound.get_length() * 1000))
            except Exception as e:
                log(f"Text-to-speech error: {e}")

        except Exception as e:
            log(f"Main loop error: {e}")

        if stop_event.is_set():
            break

    log("Listening thread stopped")

@lana_ai.route('/')
def index():
    """Render main page"""
    return render_template('lana.html')

@lana_ai.route('/start_listening', methods=['POST'])
```

```
def start_listening():
    """Start the listening process"""
    global is_listening, thread, latest_transcription, latest_response, stop_event
    if not is_listening:
        is_listening = True
        latest_transcription = ""
        latest_response = ""
        stop_event.clear()
        thread = threading.Thread(target=listen_and_respond)
        thread.start()
        return jsonify({"status": "success", "message": "Listening started"})
    return jsonify({"status": "error", "message": "Already listening"})

@lana_ai.route('/stop_listening', methods=['POST'])
def stop_listening():
    """Stop the listening process"""
    global is_listening, stop_event, thread
    if is_listening:
        is_listening = False
        stop_event.set()
        if thread:
            thread.join(timeout=5)
        return jsonify({"status": "success", "message": "Listening stopped"})
    return jsonify({"status": "error", "message": "Not currently listening"})

@lana_ai.route('/process_audio', methods=['POST'])
def process_audio():
    """Process and return audio results"""
    global latest_transcription, latest_response, audio_data
```

```
    with conversation_lock:
        if latest_transcription or latest_response:
            response = {
                "status": "success",
                "user_transcript": latest_transcription,
                "response": latest_response,
                "audio_data": audio_data.tolist()
            }
            latest_transcription = ""
            latest_response = ""
            return jsonify(response)
    return jsonify({"status": "error", "message": "No new transcription available"})

# Create required directories
os.makedirs("audio", exist_ok=True)

if __name__ == '__main__':
    app.register_blueprint(lana_ai, url_prefix='/lana_ai')
    app.run(debug=True)
```

Explanation:

I'll provide a well-structured explanation of the `lana_ai` blueprint code, which implements a voice assistant using Google's Gemini AI and speech recognition.

1) Packages - Usage

```python

import os

from flask import Flask, render_template, jsonify, Blueprint

import speech_recognition as sr

import pygame

from gtts import gTTS

from time import time

import threading

import signal

import numpy as np

from typing import Optional, Tuple

```

- **Flask Components: For creating web routes and API endpoints**

- **speech_recognition (sr): Handles speech-to-text conversion**

- **pygame: Used for audio playback**

- **gTTS (Google Text-to-Speech): Converts text responses to speech**

- **threading: Enables concurrent processing of speech recognition**

- **numpy: Handles audio data representation for visualization**

- **typing: Provides type hints for better code documentation**

- **os/signal/time: Handle file operations, process signals, and timestamps**

2) Classes and Functions - Usage and Explanation

Blueprint and API Setup

```python

lana_ai = Blueprint('lana_ai', __name__, template_folder='templates')

GOOGLE_API_KEY = 'AIzaSyBfsdtO0ipw31w1kLbuUn5_zpHZCJeLXI8'

```

- Creates a Flask Blueprint named 'lana_ai' with its own template folder

- Defines Google API key for Gemini (should be in environment variables)

AI Model Initialization

```python
from google.generativeai import configure, GenerativeModel
configure(api_key=GOOGLE_API_KEY)
model = GenerativeModel('gemini-pro')
pygame.mixer.init()
```

- Configures Google's Gemini API with the provided key
- Initializes the Gemini Pro model for text generation
- Sets up pygame audio mixer for playback

Constants and Global Variables

```python
RECORDING_PATH = "audio/recording.wav"
RESPONSE_PATH = "audio/response.mp3"
PROMPT_TEMPLATE = "You are Lana, Boss human assistant. You are witty and full of personality. Your answers should be limited to 3 lines short sentences.\nBoss: {user_input}\nLana: "
```

- Defines paths for audio files
- Creates a prompt template that gives Gemini a specific persona

```python
is_listening = False
thread = None
latest_transcription = ""
latest_response = ""
conversation_lock = threading.Lock()
stop_event = threading.Event()
audio_data = np.array([])
```

- Global state variables to track the assistant's status

- Thread synchronization primitives (lock and event)

- Storage for audio visualization data

Utility Functions

```python
def log(message: str):
    """Print and write to status.txt"""
```

- Logs messages to both console and a status file

- Used for debugging and tracking assistant activity

```python
def request_gemini(prompt: str) -> str:
    """Generate content using the Gemini model"""
```

- Sends prompts to Gemini API and returns the response

- Includes error handling to provide fallback responses

Audio Recording and Processing

```python
def enhanced_record_audio(timeout: int = 5, phrase_timeout: int = 10) -> Tuple[Optional[str], np.ndarray]:
    """Enhanced audio recording with better error handling and audio level monitoring."""
```

- Core function that handles microphone input

- Configures speech recognition parameters:

- `energy_threshold`: Determines sensitivity to speech
- `dynamic_energy_threshold`: Automatically adjusts to ambient noise
- `pause_threshold`: Controls how long to wait after speech ends

- Captures audio data for both transcription and visualization
- Returns both transcribed text and audio sample data
- Implements multiple recognition attempts with different settings

```python
def record_audio() -> str:
    """Main recording function"""
```

- Wrapper function that calls enhanced_record_audio
- Updates global audio data for visualization
- Returns status message

```python
def transcribe_audio() -> str:
    """Transcribe saved audio file using Google's speech recognition"""
```

- Secondary transcription method that processes saved audio files
- Provides another attempt at transcription if real-time fails

Main Assistant Loop

```python
def listen_and_respond():
    """Main loop for listening and responding"""
```

- Core function that runs in a separate thread

- Implements the complete voice assistant workflow:
  1. Records audio input
  2. Transcribes speech to text
  3. Generates AI response with Gemini
  4. Converts response to speech with gTTS
  5. Plays audio response
- Uses thread-safe operations to update global state
- Continues until stop_event is set

3) Routes - Usage and Explanation

```python
@lana_ai.route('/')
def index():
    """Render main page"""
    return render_template('lana.html')
```

- Root route that renders the main lana.html template
- Serves as the user interface for the voice assistant

```python
@lana_ai.route('/start_listening', methods=['POST'])
def start_listening():
    """Start the listening process"""
```

- API endpoint to activate the voice assistant
- Creates and starts a new thread for background processing
- Resets conversation state
- Returns success/error status as JSON

```python
@lana_ai.route('/stop_listening', methods=['POST'])
def stop_listening():
    """Stop the listening process"""
```

- API endpoint to deactivate the voice assistant
- Sets stop_event to signal thread termination
- Waits for thread to complete (with timeout)
- Returns success/error status as JSON

```python
@lana_ai.route('/process_audio', methods=['POST'])
def process_audio():
    """Process and return audio results"""
```

- API endpoint that retrieves the latest conversation state
- Returns:
  - User transcription (what was recognized)
  - AI response text
  - Audio visualization data
- Uses thread lock for safe access to shared state
- Clears state after returning it (to prevent duplication)

4) Other Details - Clear Explanation

Thread Management

- The code uses threading to handle audio processing in the background
- The main Flask server remains responsive while audio processing occurs

- Thread synchronization prevents race conditions:

- `conversation_lock` protects shared variables

- `stop_event` signals thread termination

Audio Visualization

- Records raw audio data as numpy arrays

- Makes this data available to frontend for visualization

- Enables real-time audio level monitoring

Error Handling Strategy

- Multiple layers of error handling:

- Try/except blocks around microphone operations

- Fallback transcription methods

- Default responses when AI generation fails

- Detailed logging throughout the process

Voice Assistant Personality

- Uses a specific prompt template to give Lana a consistent personality

- Constrains responses to 3 lines to keep interactions brief

- Described as "witty and full of personality"

Audio Processing Optimizations

- Ambient noise calibration (adjust_for_ambient_noise)

- Dynamic energy threshold adjustment

- Multiple transcription attempts with different settings

- Reasonable timeouts to prevent hanging

5) Recap of the Code

This Flask blueprint implements "Lana," a voice-based AI assistant with the following key features:

1. **Speech Recognition: Captures and transcribes user speech using Google's speech recognition**

2. **AI Response Generation: Uses Google's Gemini Pro model to generate contextual responses**

3. **Voice Synthesis: Converts text responses to spoken audio using Google TTS**

4. **Web Interface: Provides browser-based controls and visualization**

5. **Background Processing: Handles audio processing in a separate thread for responsiveness**

The voice assistant workflow:

1. User activates the assistant through the web interface

2. System listens for speech input via microphone

3. Speech is converted to text and sent to Gemini AI

4. AI generates a contextual response with a specific personality

5. Response is converted to speech and played back

6. Audio visualization data is provided to the frontend

Notable technical aspects:

- Thread-safe state management

- Comprehensive error handling

- Audio level visualization capabilities

- Rate limiting through timeouts

- Configurable speech recognition parameters

Potential improvements:

- Moving API keys to environment variables

- Adding user authentication

- Supporting conversation history/context

- Implementing wake word detection

- Adding more voice/language options

This implementation provides a solid foundation for a web-based voice assistant that combines speech recognition with generative AI to create a conversational experience.

3) INFOCRYPT:

CODE:

```
from flask import Blueprint, request, jsonify, render_template
import hashlib
import os
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.asymmetric import rsa, padding as asym_padding
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.fernet import Fernet

# Create a blueprint
infocrypt = Blueprint('infocrypt', __name__, template_folder='templates')

@infocrypt.route('/')
def index():
    return render_template('infocrypt.html')

def hash_data(data, algorithm):
    # Previous hash_data function remains the same
```

```
    try:
        if algorithm == 'CRC32':
            return format(hashlib.new('crc32', data.encode()).digest(), 'x')
        elif algorithm == 'SHA-256':
            return hashlib.sha256(data.encode()).hexdigest()
        elif algorithm == 'SHA-1':
            return hashlib.sha1(data.encode()).hexdigest()
        elif algorithm == 'SHA3-256':
            return hashlib.sha3_256(data.encode()).hexdigest()
        elif algorithm == 'BLAKE2b':
            return hashlib.blake2b(data.encode()).hexdigest()
        elif algorithm == 'SHAKE-128':
            h = hashlib.shake_128(data.encode())
            return h.digest(32).hex()
        elif algorithm == 'SHA-512':
            return hashlib.sha512(data.encode()).hexdigest()
        elif algorithm == 'SHA-384':
            return hashlib.sha384(data.encode()).hexdigest()
        else:
            return None
    except Exception as e:
        return f"Hashing Error: {str(e)}"

def encrypt_data(data, algorithm, key=None):
    try:
        if algorithm in ['AES-128', 'AES-256']:
            if key:
                # Use provided key
                key = key.encode()
```

```
            if algorithm == 'AES-128':
                key = key[:16].ljust(16, b'\0')  # Pad to 16 bytes
            else:
                key = key[:32].ljust(32, b'\0')  # Pad to 32 bytes
        else:
            # Generate random key
            key_size = 16 if algorithm == 'AES-128' else 32
            key = os.urandom(key_size)

        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
        encryptor = cipher.encryptor()
        padder = padding.PKCS7(algorithms.AES.block_size).padder()
        padded_data = padder.update(data.encode()) + padder.finalize()
        encrypted = encryptor.update(padded_data) + encryptor.finalize()
        return base64.b64encode(iv + key + encrypted).decode()

    elif algorithm == 'ChaCha20':
        if key:
            key = key.encode()[:32].ljust(32, b'\0')  # Pad to 32 bytes
        else:
            key = os.urandom(32)
        nonce = os.urandom(16)
        cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None,
backend=default_backend())
        encryptor = cipher.encryptor()
        encrypted = encryptor.update(data.encode()) + encryptor.finalize()
        return base64.b64encode(nonce + key + encrypted).decode()
```

```
    elif algorithm == 'Fernet':
        if key:
            try:
                # Try to use provided key
                key_bytes = base64.b64decode(key)
                if len(key_bytes) != 32:
                    raise ValueError("Invalid key length")
                f = Fernet(key.encode())
            except:
                # If key is invalid, generate new one
                key = Fernet.generate_key()
                f = Fernet(key)
        else:
            key = Fernet.generate_key()
            f = Fernet(key)

        encrypted = f.encrypt(data.encode())
        return base64.b64encode(key + encrypted).decode()

    elif algorithm == 'RSA':
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        public_key = private_key.public_key()
        encrypted = public_key.encrypt(
            data.encode(),
            asym_padding.OAEP(
```

```
            mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
          )
        )
        private_pem = private_key.private_bytes(
          encoding=serialization.Encoding.PEM,
          format=serialization.PrivateFormat.PKCS8,
          encryption_algorithm=serialization.NoEncryption()
        )
        private_pem_b64 = base64.b64encode(private_pem).decode()
        encrypted_b64 = base64.b64encode(encrypted).decode()
        return private_pem_b64 + ":" + encrypted_b64
      else:
        return None
    except Exception as e:
      return f"Encryption Error: {str(e)}"

def decrypt_data(encrypted_data, algorithm, key=None):
    try:
      if algorithm in ['AES-128', 'AES-256']:
        decoded_data = base64.b64decode(encrypted_data)
        key_size = 16 if algorithm == 'AES-128' else 32
        iv = decoded_data[:16]
        if key:
          key = key.encode()[:key_size].ljust(key_size, b'\0')
        else:
          key = decoded_data[16:16 + key_size]
        ciphertext = decoded_data[16 + key_size:]
```

```
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
        decryptor = cipher.decryptor()
        padded_data = decryptor.update(ciphertext) + decryptor.finalize()
        unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
        data = unpadder.update(padded_data) + unpadder.finalize()
        return data.decode()

    elif algorithm == 'ChaCha20':
        decoded_data = base64.b64decode(encrypted_data)
        nonce = decoded_data[:16]
        if key:
            key = key.encode()[:32].ljust(32, b'\0')
        else:
            key = decoded_data[16:48]
        ciphertext = decoded_data[48:]

        cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None,
backend=default_backend())
        decryptor = cipher.decryptor()
        decrypted = decryptor.update(ciphertext) + decryptor.finalize()
        return decrypted.decode()

    elif algorithm == 'Fernet':
        decoded_data = base64.b64decode(encrypted_data)
        if key:
            key = key.encode()
        else:
            key = decoded_data[:44]
```

```
            decoded_data = decoded_data[44:]
        f = Fernet(key)
        decrypted = f.decrypt(decoded_data)
        return decrypted.decode()

    elif algorithm == 'RSA':
        private_pem_b64, encrypted_b64 = encrypted_data.split(":")
        private_pem = base64.b64decode(private_pem_b64)
        ciphertext = base64.b64decode(encrypted_b64)

        private_key = serialization.load_pem_private_key(
            private_pem,
            password=None,
            backend=default_backend()
        )
        plaintext = private_key.decrypt(
            ciphertext,
            asym_padding.OAEP(
                mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return plaintext.decode()

    else:
        return None
except Exception as e:
    return f"Decryption Error: {str(e)}"
```

```
@infocrypt.route('/process', methods=['POST'])
def process_request():
    data = request.json.get('text')
    algorithm = request.json.get('algorithm')
    action = request.json.get('action')
    key = request.json.get('key')  # New parameter for custom key

    if not data or not algorithm or not action:
        return jsonify({'error': 'Text, algorithm, and action must be provided'}), 400

    result = None
    if action == 'hash':
        result = hash_data(data, algorithm)
    elif action == 'encrypt':
        result = encrypt_data(data, algorithm, key)
    elif action == 'decrypt':
        result = decrypt_data(data, algorithm, key)
    else:
        return jsonify({'error': 'Invalid action'}), 400

    if result is None:
        return jsonify({'error': 'Invalid algorithm or action'}), 400

    return jsonify({'result': result})

if __name__ == '__main__':
    infocrypt.run(debug=True)
```

Explanation:
I'll provide a well-structured explanation of the `infocrypt` blueprint, which implements cryptographic operations through a Flask API.

1) Packages - Usage

```python
from flask import Blueprint, request, jsonify, render_template
import hashlib
import os
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.asymmetric import rsa, padding as asym_padding
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.fernet import Fernet
```

- **Flask Components**: For creating web routes, handling requests, and returning JSON responses
- **hashlib**: Provides various hashing algorithms (SHA-256, SHA-512, etc.)
- **os**: Used for generating secure random bytes
- **base64**: For encoding binary cryptographic data in text format
- **cryptography.hazmat**: Comprehensive cryptography library providing:
  - Symmetric encryption (AES, ChaCha20)
  - Asymmetric encryption (RSA)
  - Padding schemes for block ciphers
  - Key serialization
- **Fernet**: High-level symmetric encryption that handles key management

2) Classes and Functions - Usage and Explanation

Blueprint Setup

```python
infocrypt = Blueprint('infocrypt', __name__, template_folder='templates')
```

- Creates a Flask Blueprint named 'infocrypt' with its own template folder

Hashing Function

```python
def hash_data(data, algorithm):
   # Various hashing implementations
```

- Computes cryptographic hash values for input data
- Supports multiple algorithms:
  - **CRC32**: Fast but less secure checksum algorithm
  - **SHA-256/SHA-1**: Secure Hash Algorithms (SHA-1 is considered weak)
  - **SHA3-256**: Newer SHA-3 standard
  - **BLAKE2b**: High-performance cryptographic hash
  - **SHAKE-128**: Extendable-output function
  - **SHA-512/SHA-384**: Larger variants of SHA-2 family
- Returns the hash in hexadecimal format
- Includes error handling to capture and report failures

Encryption Function

```python
def encrypt_data(data, algorithm, key=None):
   # Various encryption implementations
```

```

- Encrypts plaintext data using different algorithms
- Supports custom keys or generates secure random keys
- Implements multiple encryption schemes:

#### AES (Advanced Encryption Standard)
```python
if algorithm in ['AES-128', 'AES-256']:
   # Key handling, padding, and encryption
```
- Industry-standard symmetric encryption
- Supports both 128-bit and 256-bit key lengths
- Uses CBC (Cipher Block Chaining) mode
- Implements PKCS7 padding for block alignment
- Returns a base64-encoded string containing:
  - IV (Initialization Vector)
  - Encryption key (if generated)
  - Encrypted data

ChaCha20
```python
elif algorithm == 'ChaCha20':
   # ChaCha20 implementation
```
- Modern stream cipher alternative to AES
- Uses 32-byte keys and 16-byte nonces
- Returns encoded string with nonce + key + ciphertext

Fernet
```

```python
elif algorithm == 'Fernet':
    # Fernet implementation
```

- High-level symmetric encryption from the cryptography library
- Implements AES-128 in CBC mode with HMAC authentication
- Handles format validation and secure key generation
- Returns encoded key + encrypted data

RSA

```python
elif algorithm == 'RSA':
    # RSA implementation
```

- Public-key (asymmetric) encryption
- Generates 2048-bit RSA key pair on demand
- Uses OAEP padding with SHA-256
- Returns private key and ciphertext, separated by colon
- Private key is PEM-encoded and base64-encoded

Decryption Function

```python
def decrypt_data(encrypted_data, algorithm, key=None):
    # Various decryption implementations
```

- Reverses the encryption process for each algorithm
- Extracts keys and IVs from the encrypted data if not provided
- Implements appropriate padding removal
- Returns the original plaintext or error message

3) Routes - Usage and Explanation

```python
@infocrypt.route('/')
def index():
   return render_template('infocrypt.html')
```

- Root route that renders the main infocrypt.html template
- Serves as the user interface for cryptographic operations

```python
@infocrypt.route('/process', methods=['POST'])
def process_request():
   # Request processing logic
```

- Main API endpoint that handles all cryptographic operations
- Accepts JSON requests with:
  - `text`: Input data to process
  - `algorithm`: Cryptographic algorithm to use
  - `action`: Operation to perform (hash/encrypt/decrypt)
  - `key`: Optional custom key for encryption/decryption
- Validates request parameters
- Calls appropriate function based on the requested action
- Returns JSON response with result or error message

4) Other Details - Clear Explanation

Security Features

- **Secure Random Generation**: Uses `os.urandom()` for cryptographically secure random data

- **Modern Algorithms**: Implements current cryptographic standards

- **Padding**: Properly handles block cipher padding requirements

- **Error Handling**: Captures and reports cryptographic exceptions without exposing internals

Key Management

- **Custom Keys**: Supports user-provided keys where appropriate

- **Automatic Generation**: Creates secure random keys when none provided

- **Key Storage**: Embeds generated keys in the response for later decryption

- **Key Validation**: For formats like Fernet that require specific key formats

Algorithm Selection

The code offers a thoughtful selection of algorithms:

- **Hashing**: Multiple options from simple (CRC32) to complex (SHA3, BLAKE2b)

- **Symmetric Encryption**: Both traditional (AES) and modern (ChaCha20) options

- **Authenticated Encryption**: Fernet for simpler, more secure implementation

- **Asymmetric Encryption**: RSA for public-key operations

Error Handling

- Comprehensive try/except blocks around cryptographic operations

- Descriptive error messages that identify the operation that failed

- HTTP 400 responses for invalid requests

5) Recap of the Code

This Flask blueprint implements a comprehensive cryptographic toolkit with the following key features:

1. **Versatile Hashing**: Eight different hashing algorithms with varying security properties

2. **Symmetric Encryption**: Multiple options including AES (128/256-bit), ChaCha20, and Fernet

3. **Asymmetric Encryption**: RSA implementation with modern padding (OAEP)

4. **Flexible Key Management**: Support for both user-provided and auto-generated keys

5. **Browser-Based Interface**: Web frontend for easy access to cryptographic functions

The API is designed to be flexible, allowing three main operations:

1. **Hash**: One-way conversion of data to a fixed-length digest

2. **Encrypt**: Convert plaintext to ciphertext with various algorithms

3. **Decrypt**: Restore original plaintext from ciphertext

Technical highlights:

- Proper implementation of initialization vectors (IVs) and nonces

- Appropriate padding mechanisms for block ciphers

- Secure key generation and handling

- Base64 encoding for binary cryptographic data

Potential improvements:

- Moving to environment variables for any sensitive values

- Adding authentication to restrict API access

- Implementing key derivation functions for password-based encryption

- Adding digital signature capabilities

- Supporting additional modes like GCM for authenticated encryption

The implementation provides a solid foundation for cryptographic operations that could be integrated into various security-focused applications, from password management to secure messaging.

4) FileFender:

CODE:

```
import os
from flask import Flask, Blueprint, request, render_template, jsonify
from werkzeug.utils import secure_filename
import requests
import time
import logging

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = Flask(__name__)

VIRUSTOTAL_API_KEY = '4b4ae68cf38ed487342818091ad6ea879d11207e57049616f55fcd5c869233f9'
VIRUSTOTAL_API_URL = 'https://www.virustotal.com/api/v3'

filescanner = Blueprint('filescanner', __name__, template_folder='templates')

@filescanner.route('/')
def index():
    return render_template('filescanner.html')

@filescanner.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        return jsonify({'error': 'No file part'}), 400
```

```
    file = request.files['file']
    if file.filename == '':
        return jsonify({'error': 'No selected file'}), 400
    if file:
        filename = secure_filename(file.filename)
        temp_dir = os.path.join(os.path.dirname(__file__), 'temp')
        os.makedirs(temp_dir, exist_ok=True)
        file_path = os.path.join(temp_dir, filename)
        file.save(file_path)
        try:
            result = scan_file(file_path)
            os.remove(file_path)
            return jsonify(result)
        except TimeoutError as e:
            os.remove(file_path)
            logger.warning(f"Scan timeout for file: {filename}")
            return jsonify({'error': 'Scan is taking longer than expected. Please try again later.'}),
202
        except Exception as e:
            if os.path.exists(file_path):
                os.remove(file_path)
            logger.error(f"Error during file scan: {str(e)}")
            return jsonify({'error': 'An unexpected error occurred during the scan.'}), 500


def scan_file(file_path):
    url = f'{VIRUSTOTAL_API_URL}/files'
    headers = {'x-apikey': VIRUSTOTAL_API_KEY}


    with open(file_path, 'rb') as file:
        files = {'file': (os.path.basename(file_path), file)}
```

```
        response = requests.post(url, headers=headers, files=files)

    response.raise_for_status()
    upload_result = response.json()

    if 'data' not in upload_result or 'id' not in upload_result['data']:
        raise ValueError('Failed to upload file to VirusTotal')

    analysis_id = upload_result['data']['id']
    return get_analysis_result(analysis_id)

def get_analysis_result(analysis_id):
    url = f'{VIRUSTOTAL_API_URL}/analyses/{analysis_id}'
    headers = {'x-apikey': VIRUSTOTAL_API_KEY}

    max_attempts = 20  # Increased from 10
    wait_time = 10  # Increased from 5

    for attempt in range(max_attempts):
        try:
            response = requests.get(url, headers=headers)
            response.raise_for_status()
            result = response.json()

            if result['data']['attributes']['status'] == 'completed':
                return process_analysis_result(result)

            logger.info(f"Scan in progress. Attempt {attempt + 1}/{max_attempts}")
            time.sleep(wait_time)
```

```
            except requests.RequestException as e:
                logger.error(f"Error getting analysis result (attempt {attempt + 1}): {str(e)}")

        raise TimeoutError('Analysis timed out')

def process_analysis_result(result):
    stats = result['data']['attributes']['stats']
    total_scans = sum(stats.values())
    malicious = stats.get('malicious', 0)
    suspicious = stats.get('suspicious', 0)

    risk_score = (malicious + suspicious) / total_scans * 100 if total_scans > 0 else 0

    return {
        'risk_score': round(risk_score, 2),
        'total_scans': total_scans,
        'malicious': malicious,
        'suspicious': suspicious,
        'full_result': result
    }
# Register the blueprint
app.register_blueprint(filescanner, url_prefix='/filescanner')

if __name__ == '__main__':
    # Ensure temp directory exists
    temp_dir = os.path.join(os.path.dirname(__file__), 'temp')
    os.makedirs(temp_dir, exist_ok=True)

    app.run(debug=True)
```

Explanation:

I'll analyze your Flask backend code that integrates with VirusTotal for file scanning. Here's a comprehensive breakdown:

1) Packages - Usage

Core Packages:

- `os`: Handles file paths, directory creation, and file operations
- `flask`: Web framework for creating the application and routes
- `werkzeug.utils`: Provides `secure_filename` to sanitize uploaded filenames
- `requests`: Makes HTTP requests to the VirusTotal API
- `time`: Implements waiting periods between API polling attempts
- `logging`: Sets up application logging for monitoring and debugging

2) Classes and Functions - Usage and Explanation

Application Setup:

- `Flask(__name__)`: Creates the main Flask application instance
- `Blueprint('filescanner',...)`: Creates a modular component that can be registered with the main app

File Scanning Functions:

- `scan_file(file_path)`:
  - Uploads a file to VirusTotal's API
  - Returns the analysis results
  - Handles file opening, POST request creation, and response processing

- `get_analysis_result(analysis_id)`:
  - Polls the VirusTotal API to check scan status

- Implements retry logic with exponential backoff (20 attempts, 10 seconds between)
- Returns completed scan results or raises timeout exception

- `process_analysis_result(result)`:
- Extracts and calculates key metrics from scan results
- Computes a risk score as percentage: (malicious + suspicious) / total_scans * 100
- Returns a structured response with risk metrics and full scan details

3) Routes - Usage and Explanation

`@filescanner.route('/')`
- Renders the main file scanner interface template
- Entry point for users to interact with the scanning service

`@filescanner.route('/upload', methods=['POST'])`
- Handles file upload submission
- Validates presence of file in request
- Secures filename to prevent path traversal attacks
- Creates temporary storage for the uploaded file
- Initiates scan process and returns results
- Implements comprehensive error handling and cleanup

4) Other Details

API Configuration:
- `VIRUSTOTAL_API_KEY`: Authentication key for VirusTotal API access
- `VIRUSTOTAL_API_URL`: Base URL for API endpoints

Error Handling:

- Specific error handling for missing files, timeouts, and unexpected errors
- Proper HTTP status codes returned (400, 202, 500)
- File cleanup in all error scenarios

Security Considerations:
- Uses `secure_filename()` to prevent path traversal attacks
- Implements temporary file storage with cleanup
- Error messages are informative but don't expose system details

Performance Optimization:
- Increased polling attempts (20) and wait time (10s) for large files
- Asynchronous response pattern for long-running scans

5) Recap of the Code

This is a Flask-based file scanning service that:

1. Provides a web interface for file uploads
2. Securely handles uploaded files with proper validation
3. Integrates with VirusTotal's API to scan files for malware
4. Implements robust error handling and timeout management
5. Calculates a risk score based on scan results
6. Returns structured scan reports to the client
7. Uses a Blueprint architecture for modularity and scalability

The application follows good practices for security (secure file handling), resource management (file cleanup), and error handling. It effectively abstracts the complexity of the VirusTotal API behind a simple interface while providing detailed scan results.

The code is designed to handle timeouts gracefully, which is important since virus scanning can be time-intensive for larger files. It also implements proper logging to facilitate debugging and monitoring in production.

5) PORTSCANNER:

CODE:

```
from flask import Flask, request, jsonify, render_template, Blueprint
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
import nmap
import socket
from datetime import datetime
import concurrent.futures
import os
import validators
from urllib.parse import urlparse
from functools import wraps

portscanner = Blueprint('portscanner', __name__, url_prefix='/portscanner')

class Config:
    SCAN_TIMEOUT = 6000  # 10 minutes for longer scans
    MAX_CONCURRENT_SCANS = 5
    DEFAULT_PORTS = '1-1024'

    SCAN_TYPES = {
        'intense_scan': {'command': '-T4 -A -v -Pn', 'description': 'Comprehensive scan'},
        'service_version': {'command': '-sV -Pn', 'description': 'Service version detection'},
        'os_detection': {'command': '-O -Pn', 'description': 'OS detection'},
        'tcp_connect': {'command': '-sT -Pn', 'description': 'TCP connect scan'},
        'syn_scan': {'command': '-sS -Pn', 'description': 'SYN scan (requires root)'},
        'udp_scan': {'command': '-sU -Pn', 'description': 'UDP scan'},
```

```
    'aggressive_scan': {'command': '-A -Pn', 'description': 'Aggressive scan with version detection'},
    'list_scan': {'command': '-sL -Pn', 'description': 'Lists targets without scanning'},
    'null_scan': {'command': '-sN -Pn', 'description': 'Null scan with no flags'},
    'xmas_scan': {'command': '-sX -Pn', 'description': 'Xmas scan with FIN, PSH, URG flags'},
    'fin_scan': {'command': '-sF -Pn', 'description': 'FIN scan with only FIN flag'},
    'full_port_scan': {'command': '-p- -Pn', 'description': 'Scans all 65,535 ports'},
    'script_scan': {'command': '-sC -Pn', 'description': 'Default script scan'},
    'version_intensity': {'command': '--version-intensity 9 -Pn', 'description': 'Intense version detection'},
    'timing_aggressive': {'command': '-T4 -Pn', 'description': 'Aggressive timing template'},
    'timing_insane': {'command': '-T5 -Pn', 'description': 'Insane timing template'},
    'traceroute': {'command': '--traceroute -Pn', 'description': 'Trace path to host'},
    'fragment_scan': {'command': '-f -Pn', 'description': 'Fragment packets'},
    'idle_scan': {'command': '-sI -Pn', 'description': 'Idle scan'},
    'ack_scan': {'command': '-sA -Pn', 'description': 'ACK scan'},
    'window_scan': {'command': '-sW -Pn', 'description': 'Window scan'},
    'maimon_scan': {'command': '-sM -Pn', 'description': 'Maimon scan'},
    'sctp_init_scan': {'command': '-sY -Pn', 'description': 'SCTP INIT scan'},
    'sctp_cookie_scan': {'command': '-sZ -Pn', 'description': 'SCTP COOKIE-ECHO scan'}
  }

# Configure rate limiting
limiter = Limiter(
  key_func=get_remote_address,
  default_limits=["100 per day", "10 per minute"]
)

def requires_root(f):
```

```
    @wraps(f)
    def decorated_function(*args, **kwargs):
        root_required_scans = ['syn_scan']
        scan_type = request.json.get('scanType')
        if scan_type in root_required_scans and os.geteuid() != 0:
            return jsonify({
                'error': f'The {scan_type} requires root privileges',
                'success': False
            }), 403
        return f(*args, **kwargs)
    return decorated_function

class PortScanner:
    def __init__(self):
        self.nmap = nmap.PortScanner()
        self.scan_pool = concurrent.futures.ThreadPoolExecutor(
            max_workers=Config.MAX_CONCURRENT_SCANS
        )

    def validate_target(self, target):
        """Validate and normalize target input (supports IP, domain, and URL)"""
        try:
            # Remove protocol and path if URL
            if '//' in target:
                parsed = urlparse(target)
                target = parsed.netloc or parsed.path

            # Remove any remaining path components and query parameters
            target = target.split('/')[0].split('?')[0]
```

```
        # Check if IP address
        try:
            socket.inet_aton(target)
            return target, None
        except socket.error:
            pass

        # Check if domain
        if validators.domain(target) or target.startswith('localhost'):
            try:
                socket.gethostbyname(target)
                return target, None
            except socket.gaierror:
                return None, "Domain cannot be resolved"

        return None, "Invalid target format. Please provide a valid IP address, domain, or
URL."

    except Exception as e:
        return None, f"Validation error: {str(e)}"

  def perform_port_scan(self, target, scan_type):
    """Perform Nmap port scan with error handling"""
    try:
        if scan_type not in Config.SCAN_TYPES:
            return {"error": "Invalid scan type", "success": False}

        scan_args = Config.SCAN_TYPES[scan_type]['command']
```

```
# Adjust ports based on scan type
if scan_type == 'full_port_scan':
    ports = '1-65535'
elif scan_type == 'quick_scan':
    ports = 'T:21-25,80,139,443,445,3389'  # Common ports
else:
    ports = Config.DEFAULT_PORTS

try:
    scan_results = self.nmap.scan(
        hosts=target,
        ports=ports,
        arguments=scan_args,
        timeout=Config.SCAN_TIMEOUT
    )

    if not scan_results or 'scan' not in scan_results:
        return {"error": "Scan produced no results", "success": False}

except nmap.PortScannerError as e:
    return {"error": f"Scan failed: {str(e)}", "success": False}
except Exception as e:
    return {"error": f"Unexpected error: {str(e)}", "success": False}

processed_results = {
    'scan_info': {
        'target': target,
        'scan_type': scan_type,
        'description': Config.SCAN_TYPES[scan_type]['description'],
```

```
            'command_used': f"nmap {scan_args} -p {ports} {target}",
            'start_time': datetime.now().isoformat(),
            'elapsed': self.nmap.scanstats().get('elapsed', '0'),
            'total_hosts': self.nmap.scanstats().get('totalhosts', '0'),
            'up_hosts': self.nmap.scanstats().get('uphosts', '0'),
            'down_hosts': self.nmap.scanstats().get('downhosts', '0')
        },
        'hosts': {},
        'success': True
    }

    for host in self.nmap.all_hosts():
        host_data = {
            'state': self.nmap[host].state(),
            'protocols': {},
            'hostnames': self.nmap[host].hostnames()
        }

        # Add OS detection results if available
        if hasattr(self.nmap[host], 'osmatch') and self.nmap[host].osmatch():
            host_data['os_matches'] = self.nmap[host].osmatch()

        # Add traceroute if available
        if hasattr(self.nmap[host], 'traceroute'):
            host_data['traceroute'] = self.nmap[host].traceroute()

        # Process each protocol
        for proto in self.nmap[host].all_protocols():
            ports = self.nmap[host][proto].keys()
```

```
                host_data['protocols'][proto] = {}

                for port in ports:
                    port_info = self.nmap[host][proto][port]
                    port_data = {
                        'state': port_info.get('state'),
                        'service': port_info.get('name'),
                        'product': port_info.get('product', ''),
                        'version': port_info.get('version', ''),
                        'extrainfo': port_info.get('extrainfo', ''),
                        'reason': port_info.get('reason', ''),
                        'cpe': port_info.get('cpe', [])
                    }

                    # Add script output if available
                    if 'script' in port_info:
                        port_data['scripts'] = port_info['script']

                    host_data['protocols'][proto][port] = port_data

            processed_results['hosts'][host] = host_data

        return processed_results

    except Exception as e:
        return {"error": str(e), "success": False}

# Initialize scanner
scanner = PortScanner()
```

```
@portscanner.route('/')
def index():
    return render_template('portscanner.html')

@portscanner.route('/api/scan-types', methods=['GET'])
def get_scan_types():
    """Return available scan types and their descriptions"""
    return jsonify({
        'scan_types': Config.SCAN_TYPES,
        'success': True
    })

@portscanner.route('/api/scan', methods=['POST'])
@limiter.limit("5 per minute")
@requires_root
def port_scan():
    """Port scanner endpoint"""
    try:
        data = request.get_json()
        if not data:
            return jsonify({"error": "No data provided", "success": False}), 400

        target = data.get('target', '').strip()
        scan_type = data.get('scanType', 'quick_scan')

        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400
```

```
        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        if scan_type not in Config.SCAN_TYPES:
            return jsonify({"error": "Invalid scan type", "success": False}), 400

        results = scanner.perform_port_scan(normalized_target, scan_type)
        if not results.get('success', False):
            return jsonify(results), 400

        return jsonify(results)

    except Exception as e:
        return jsonify({"error": str(e), "success": False}), 500

@portscanner.errorhandler(429)
def ratelimit_handler(e):
    return jsonify({"error": "Rate limit exceeded", "success": False}), 429

def create_app():
    app = Flask(__name__)
    app.register_blueprint(portscanner)
    limiter.init_app(app)
    return app

if __name__ == '__main__':
    app = create_app()
    app.run(debug=False, host='0.0.0.0', port=5000)
```

Explanation:
I'll provide a detailed analysis of your port scanner backend code:

1) Packages - Usage

Core Packages:

- `flask`: Web framework for creating the API and routes
- `flask_limiter`: Implements rate limiting to prevent API abuse
- `nmap`: Python wrapper for the Nmap port scanning tool
- `socket`: Handles low-level network operations and validation
- `concurrent.futures`: Manages concurrent scan operations
- `validators`: Validates domain format
- `urllib.parse`: Extracts components from URLs
- `datetime`: Timestamps scan operations
- `os`: Checks for root privileges
- `functools.wraps`: Creates proper decorator functions

2) Classes and Functions - Usage and Explanation

`Config` Class:

- Static configuration for the scanner application
- Defines scan timeout, concurrency limits, and default ports
- Contains comprehensive dictionary of scan types with command arguments and descriptions

`requires_root` Decorator:

- Checks if privileged scan types are being requested
- Verifies root access using `os.geteuid()`
- Returns appropriate error response if privileges are insufficient

`PortScanner` Class:

- Core scanning functionality encapsulation
- `__init__`: Initializes nmap scanner and ThreadPoolExecutor
- `validate_target`:
  - Normalizes and validates input (IP, domain, URL)
  - Handles protocol stripping and domain resolution
  - Returns normalized target or error message
- `perform_port_scan`:
  - Executes nmap scan with specified parameters
  - Processes raw nmap results into structured JSON
  - Handles multiple exception scenarios
  - Extracts comprehensive scan data (OS detection, services, versions)

3) Routes - Usage and Explanation

`@portscanner.route('/')`

- Renders the main port scanner interface template
- Entry point for the web application

`@portscanner.route('/api/scan-types', methods=['GET'])`

- Returns all available scan types with descriptions
- Allows frontend to dynamically populate scan options

`@portscanner.route('/api/scan', methods=['POST'])`

- Main scanning endpoint with rate limiting (5 requests per minute)
- Root privilege verification for certain scan types
- Input validation and normalization
- Executes scan and returns structured results
- Comprehensive error handling

**`@portscanner.errorhandler(429)` **

- Custom handler for rate limit exceeded errors
- Returns user-friendly JSON response

4) Other Details

Security Features:

- Rate limiting to prevent DoS attacks
- Input validation for target hostnames/IPs
- Privilege checking for advanced scan types
- Error handling that doesn't expose system details

Concurrency Management:

- ThreadPoolExecutor limits concurrent scans to prevent resource exhaustion
- Configurable maximum concurrent scans (default: 5)

Scan Flexibility:

- 24 different scan types with various detection capabilities
- Customizable port ranges based on scan type
- Detailed output including service versions, OS detection, scripts

Architecture:

- Blueprint-based modular design for easy integration
- Factory pattern with `create_app()` for application creation
- Separation of concerns between configuration, scanning logic, and API routes

5) Recap of the Code

This is a comprehensive Flask-based port scanning API that:

1. Provides a web interface for network scanning

2. Supports multiple scan types with different capabilities and intensity levels

3. Implements robust security measures (rate limiting, input validation, privilege checks)

4. Uses concurrent execution for better performance

5. Returns detailed, structured scan results including service detection and OS fingerprinting

6. Handles edge cases and errors gracefully

The code follows security best practices by validating all inputs, limiting scan frequency, checking for required privileges, and providing detailed error messages without exposing system internals.

It's designed as a Blueprint for modular integration into larger applications while maintaining clean separation of concerns. The comprehensive scan type definitions allow for flexibility in scanning approaches, from quick reconnaissance to detailed service enumeration.

The error handling is particularly robust, with specific responses for different failure scenarios and proper HTTP status codes.

6) SNAPSPEAK_AI:

CODE:

```
from flask import Blueprint, request, jsonify, render_template, current_app
from flask_cors import CORS
from transformers import BlipForConditionalGeneration, BlipProcessor, logging
import torch
from PIL import Image, ExifTags
from PIL.ExifTags import TAGS, GPSTAGS
```

```
import io
import time
import imagehash
import traceback
import warnings
import cv2
import numpy as np
import binascii
from sklearn.cluster import KMeans
import hashlib
import os
from collections import Counter
import requests

# Suppress warnings
warnings.filterwarnings("ignore", category=FutureWarning)
logging.set_verbosity_error()

# Blueprint setup
snapspeak_ai = Blueprint('snapspeak_ai', __name__, template_folder='templates')
CORS(snapspeak_ai)

# Global configurations
FACE_DETECTION_CONFIDENCE_THRESHOLD = 0.85
COLOR_CLUSTER_COUNT = 5
IMAGE_RESIZE_DIMENSION = 150
DARK_PIXEL_THRESHOLD = 10

# Initialize models
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-image-captioning-large").to(device)
processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-large")

# Initialize face detection
try:
    from deepface import DeepFace
    DEEPFACE_AVAILABLE = True
except ImportError:
    DEEPFACE_AVAILABLE = False
    print("Warning: DeepFace not available")

def get_labeled_gps(gps_info):
    """Convert GPS information to readable format"""
    labeled = {}
    for key, value in gps_info.items():
        if key in GPSTAGS:
            tag_name = GPSTAGS[key]
            if isinstance(value, tuple) and len(value) > 0:
                if tag_name in ['GPSLatitude', 'GPSLongitude']:
                    try:
                        degrees = float(value[0][0]) / float(value[0][1])
                        minutes = float(value[1][0]) / float(value[1][1])
                        seconds = float(value[2][0]) / float(value[2][1])
                        labeled[tag_name] = f"{degrees:.6f}° {minutes:.4f}' {seconds:.2f}\""
                    except:
                        labeled[tag_name] = value
                else:
                    labeled[tag_name] = value
```

```
        else:
            labeled[tag_name] = value
    return labeled

def format_binary_data(data):
    """Format binary data into readable format"""
    if isinstance(data, bytes):
        try:
            decoded = data.decode('utf-8')
            return decoded if all(32 <= ord(c) <= 126 for c in decoded) else f"HEX:
{binascii.hexlify(data).decode('ascii')}"
        except UnicodeDecodeError:
            return f"HEX: {binascii.hexlify(data).decode('ascii')}"
    return str(data)

def metadata_analysis(image):
    """Extract and analyze image metadata"""
    try:
        exif_data = {
            'Format': image.format,
            'Mode': image.mode,
            'Size': f"{image.width}x{image.height}",
            'Bits_Per_Channel': getattr(image, 'bits', 8)
        }

        # Extract EXIF data
        info = image.getexif()
        if info:
            for tag_id, value in info.items():
                tag = TAGS.get(tag_id, tag_id)
```

```
        if tag == 'GPSInfo':
            gps_data = get_labeled_gps(value)
            exif_data.update({f'GPS_{k}': v for k, v in gps_data.items()})
        else:
            exif_data[tag] = value

# Get ICC Profile information
if 'icc_profile' in image.info:
    exif_data['ICC_Profile'] = format_binary_data(image.info['icc_profile'])

# Extract quantization tables if available
if hasattr(image, "quantization"):
    exif_data['Quantization_Tables'] = image.quantization

# Add additional image info
for key, value in image.info.items():
    if key not in ['exif', 'icc_profile']:
        exif_data[key] = value

# Calculate image statistics
if image.mode == 'RGB':
    r, g, b = image.split()
    exif_data.update({
        'Entropy_R': f"{r.entropy():.2f}",
        'Entropy_G': f"{g.entropy():.2f}",
        'Entropy_B': f"{b.entropy():.2f}",
        'Color_Range': str(image.getextrema())
    })
```

```
        return {k: format_binary_data(v) for k, v in exif_data.items()}
    except Exception as e:
        print(f"Error in metadata analysis: {str(e)}")
        return {}

def enhanced_face_detection(image):
    """Enhanced face detection using DeepFace or OpenCV fallback"""
    try:
        np_image = np.array(image)

        if len(np_image.shape) == 2:
            np_image = cv2.cvtColor(np_image, cv2.COLOR_GRAY2RGB)
        elif np_image.shape[2] == 4:
            np_image = cv2.cvtColor(np_image, cv2.COLOR_RGBA2RGB)

        face_locations = []

        if DEEPFACE_AVAILABLE:
            try:
                faces = DeepFace.extract_faces(
                    np_image,
                    detector_backend='retinaface',
                    enforce_detection=False,
                    align=True
                )

                for face in faces:
                    confidence = face.get('confidence', 0)
                    if confidence > FACE_DETECTION_CONFIDENCE_THRESHOLD:
```

```
                facial_area = face['facial_area']
                face_locations.append({
                    'x': int(facial_area['x']),
                    'y': int(facial_area['y']),
                    'width': int(facial_area['w']),
                    'height': int(facial_area['h']),
                    'confidence': float(confidence),
                    'detector': 'retinaface'
                })
        except Exception as e:
            print(f"DeepFace detection failed, falling back to OpenCV: {str(e)}")
            DEEPFACE_AVAILABLE = False

    if not DEEPFACE_AVAILABLE:
        face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
        gray = cv2.cvtColor(np_image, cv2.COLOR_RGB2GRAY)

        faces = face_cascade.detectMultiScale(
            gray,
            scaleFactor=1.3,
            minNeighbors=5,
            minSize=(30, 30),
            flags=cv2.CASCADE_SCALE_IMAGE
        )

        for (x, y, w, h) in faces:
            face_locations.append({
                'x': int(x),
                'y': int(y),
```

```
                'width': int(w),
                'height': int(h),
                'confidence': 0.8,
                'detector': 'opencv'
            })

        return {
            'count': len(face_locations),
            'locations': face_locations,
            'success': True
        }

    except Exception as e:
        print(f"Error in face detection: {str(e)}")
        return {
            'count': 0,
            'locations': [],
            'success': False,
            'error': str(e)
        }

def color_analysis(image):
    """Analyze dominant colors using K-means clustering"""
    try:
        if image.mode != 'RGB':
            image = image.convert('RGB')

        thumb = image.copy()
        thumb.thumbnail((IMAGE_RESIZE_DIMENSION, IMAGE_RESIZE_DIMENSION))
```

```
        pixels = np.array(thumb).reshape(-1, 3)

        kmeans = KMeans(n_clusters=COLOR_CLUSTER_COUNT, random_state=0,
n_init=10)
        kmeans.fit(pixels)

        colors = kmeans.cluster_centers_
        labels = kmeans.labels_
        counts = np.bincount(labels)

        total_pixels = sum(counts)
        color_info = []

        sorted_indices = np.argsort(counts)[::-1]

        for idx in sorted_indices:
            r, g, b = map(int, colors[idx])
            hex_color = f'#{r:02x}{g:02x}{b:02x}'
            percentage = (counts[idx] / total_pixels) * 100

            color_info.append({
                'hex': hex_color,
                'rgb': f'rgb({r},{g},{b})',
                'percentage': round(percentage, 1)
            })

        return color_info
    except Exception as e:
        print(f"Error in color analysis: {str(e)}")
```

```
        return []

def detect_steganography(image):
    """Advanced steganography detection"""
    try:
        if image.mode != 'RGB':
            image = image.convert('RGB')

        pixels = np.array(image)
        lsb = pixels & 1

        # Calculate entropy of LSB
        lsb_entropy = cv2.calcHist([lsb.ravel()], [0], None, [2], [0, 2])
        lsb_entropy = float(sum(-p * np.log2(p + 1e-10) for p in lsb_entropy))

        # Analyze hidden pixels
        gray_image = cv2.cvtColor(pixels, cv2.COLOR_RGB2GRAY)
        hidden_pixel_count = int(np.sum(gray_image < DARK_PIXEL_THRESHOLD))

        threshold = 0.97
        confidence = min((lsb_entropy / threshold) * 100, 100)

        return {
            'detected': lsb_entropy > threshold,
            'confidence': float(confidence),
            'methods': ['LSB Analysis'] if lsb_entropy > threshold else [],
            'hidden_pixels': hidden_pixel_count,
            'hidden_pixel_threshold': DARK_PIXEL_THRESHOLD
        }
```

```
    except Exception as e:
        print(f"Error in steganography detection: {str(e)}")
        return {
            'detected': False,
            'confidence': 0,
            'methods': [],
            'hidden_pixels': 0,
            'error': str(e)
        }

@torch.no_grad()
def generate_caption(image):
    """Generate image caption using BLIP model"""
    try:
        pixel_values = processor(images=image,
return_tensors="pt").pixel_values.to(device)
        output_ids = model.generate(pixel_values, max_length=50, num_beams=4)
        return processor.decode(output_ids[0], skip_special_tokens=True)
    except Exception as e:
        print(f"Error in caption generation: {str(e)}")
        return "Error generating caption"

def generate_image_digest(image_bytes):
    """Generate SHA-256 hash of the image"""
    try:
        return hashlib.sha256(image_bytes).hexdigest()
    except Exception as e:
        print(f"Error generating image digest: {str(e)}")
        return None
```

```
def image_hash(image):
    """Generate perceptual hash"""
    return str(imagehash.average_hash(image))

@snapspeak_ai.route('/')
def index():
    return render_template('snapspeak.html')

@snapspeak_ai.route('/api/analyze/', methods=['POST'])
def analyze_image():
    """Main image analysis endpoint"""
    try:
        start_time = time.time()

        if 'file' not in request.files:
            return jsonify({'error': 'No file provided'}), 400

        file = request.files['file']
        if not file.filename:
            return jsonify({'error': 'No selected file'}), 400

        try:
            image_bytes = file.read()
            image = Image.open(io.BytesIO(image_bytes))
        except Exception as e:
            return jsonify({'error': 'Invalid image file'}), 400

        # Perform all analyses
        analysis_results = {
```

```
            'caption': generate_caption(image),
            'metadata': metadata_analysis(image),
            'image_hash': image_hash(image),
            'sha256_digest': generate_image_digest(image_bytes),
            'dominant_colors': color_analysis(image),
            'faces': enhanced_face_detection(image),
            'steganography': detect_steganography(image),
            'processing_time': float(time.time() - start_time)
        }

        return jsonify(analysis_results)

    except Exception as e:
        error_trace = traceback.format_exc()
        print(f"Error in analyze_image: {str(e)}")
        print(f"Traceback: {error_trace}")
        return jsonify({
            'error': str(e),
            'traceback': error_trace if current_app.debug else 'Enable debug mode for traceback'
        }), 500

if __name__ == "__main__":
    from flask import Flask
    app = Flask(__name__)
    app.register_blueprint(snapspeak_ai, url_prefix='/snapspeak_ai')
    app.run(debug=True)
```

EXPLANATION:

I'll analyze your image analysis backend code with a comprehensive breakdown:

1) Packages - Usage

Core Packages:

- `flask`: Web framework for building the API
- `flask_cors`: Enables Cross-Origin Resource Sharing
- `transformers`: Hugging Face library for the BLIP image captioning model
- `torch`: Deep learning framework for model execution
- `PIL` (Pillow): Core image processing operations and metadata extraction
- `cv2` (OpenCV): Face detection and image processing algorithms
- `numpy`: Array operations for image manipulation
- `imagehash`: Perceptual image hashing
- `sklearn.cluster`: K-means clustering for color analysis
- `hashlib`: Cryptographic hashing for image digests
- `deepface`: Advanced face recognition (optional dependency)

2) Classes and Functions - Usage and Explanation

Metadata Functions:

- `get_labeled_gps(gps_info)`:
  - Converts raw GPS EXIF data to human-readable format
  - Handles degree/minute/second conversion for coordinates

- `format_binary_data(data)`:
  - Makes binary metadata human-readable
  - Converts to hex representation when not UTF-8 decodable

- `metadata_analysis(image)`:
  - Extracts comprehensive metadata (format, dimensions, EXIF)
  - Processes GPS data, ICC profiles, and quantization tables
  - Calculates channel-specific entropy statistics

Analysis Functions:
- `enhanced_face_detection(image)`:
  - Detects faces using DeepFace (RetinaFace backend) with OpenCV fallback
  - Handles grayscale and alpha channel conversion
  - Returns face locations, counts, and confidence scores

- `color_analysis(image)`:
  - Identifies dominant colors using K-means clustering
  - Calculates color percentages and provides RGB/hex values
  - Optimizes performance with image thumbnail resizing

- `detect_steganography(image)`:
  - Implements LSB (Least Significant Bit) analysis for hidden data detection
  - Calculates entropy of LSBs to identify potential steganography
  - Detects unusually dark pixels that might conceal information

AI/ML Functions:
- `generate_caption(image)`:
  - Uses BLIP (Salesforce/blip-image-captioning-large) model
  - Generates natural language description of image content
  - Implements beam search for higher quality captions

Hash/Identity Functions:
- `generate_image_digest(image_bytes)`:

- Creates cryptographic SHA-256 hash of raw image data
- Useful for exact image matching and integrity verification

- `image_hash(image)`:
- Generates perceptual hash for similarity matching
- Allows finding visually similar images even after modifications

3) Routes - Usage and Explanation

`@snapspeak_ai.route('/')`
- Renders the main UI template for the image analysis tool
- Entry point for users to interact with the application

`@snapspeak_ai.route('/api/analyze/', methods=['POST'])`
- Main analysis endpoint that orchestrates the entire workflow
- Handles file upload validation and processing
- Executes all analysis functions in sequence:
1. Image captioning (AI description)
2. Metadata extraction and analysis
3. Perceptual and cryptographic hashing
4. Color analysis
5. Face detection
6. Steganography detection
- Tracks and returns processing time
- Implements comprehensive error handling

4) Other Details

Configuration and Initialization:

- Uses Blueprint pattern for modular application design
- Configurable thresholds for face detection and color analysis
- Intelligent device selection (CUDA/CPU) for model execution
- Warning suppression for cleaner logs

Error Handling:
- Comprehensive try/except blocks throughout the codebase
- Detailed error logging with traceback for debugging
- Conditional exposure of debug information based on app configuration

Performance Considerations:
- Image resizing for efficient color analysis
- Torch no_grad context for inference optimization
- Fallback mechanisms for missing dependencies

Security Features:
- Input validation for uploaded files
- Secure error handling that doesn't expose system details
- Cross-origin resource sharing configuration

5) Recap of the Code

This is a sophisticated image analysis API that:
1. Leverages AI to generate natural language descriptions of images
2. Extracts and interprets comprehensive metadata including GPS coordinates
3. Performs advanced face detection with multiple backend options
4. Analyzes color composition using machine learning clustering
5. Detects potential hidden data through steganography analysis
6. Generates both cryptographic and perceptual image hashes

7. Presents all findings through a clean REST API

The architecture follows modern best practices:

- Clear separation of concerns with specialized functions

- Comprehensive error handling and logging

- Fallback mechanisms for robust operation

- Blueprint-based modular design for easy integration

The code demonstrates advanced capabilities in:

- Computer vision (face detection, color analysis)

- Natural language processing (image captioning)

- Digital forensics (metadata analysis, steganography detection)

- Perceptual hashing (content-based image retrieval)

This backend would be particularly valuable for applications in digital forensics, content moderation, accessibility (image descriptions), or digital asset management systems where deep image understanding is required.

7) Cybersentry_AI:

CODE:

```
import json

import sys

from io import StringIO

from flask import Blueprint, render_template, request, jsonify

import google.generativeai as genai

from fuzzywuzzy import fuzz


# Create a blueprint

cybersentry_ai = Blueprint('cybersentry_ai', __name__, template_folder='templates')
```

```
# Load responses from JSON file
def load_responses():
    try:
        with open('responses.json', 'r') as file:
            return json.load(file)
    except Exception as e:
        print(f"Error loading responses: {e}")
        return []

responses = load_responses()

# Configure Gemini API
genai.configure(api_key='AIzaSyBZgKntCoDX9ofCAntTRxWTEc0N37fxlQk')
model = genai.GenerativeModel('gemini-1.5-flash-latest')

def capture_output(func):
    def wrapper(*args, **kwargs):
        old_stdout = sys.stdout
        sys.stdout = StringIO()

        result = func(*args, **kwargs)

        output = sys.stdout.getvalue()
        sys.stdout = old_stdout

        return result, output
    return wrapper
```

```
@capture_output
def fuzzy_match(query, responses, threshold=80):
    query = query.lower().strip()
    best_match = None
    best_score = 0

    for response in responses:
        if 'question' in response:
            score = fuzz.token_set_ratio(query, response['question'].lower())
            if score > best_score and score >= threshold:
                best_score = score
                best_match = response

    return best_match.get('answer') if best_match else None

@capture_output
def get_gemini_response(query):
    try:
        context = "You are a cybersecurity AI assistant. Provide accurate and helpful information about cybersecurity topics. If you're not sure about something, provide the most likely answer based on your knowledge without disclaimers."
        full_prompt = f"{context}\n\nUser: {query}\nAssistant:"
        response = model.generate_content(full_prompt)
        return response.text
    except Exception as e:
        print(f"Error fetching response from Gemini API: {e}")
        return None

@cybersentry_ai.route('/')
def index():
```

```
    return render_template('cybersentry_AI.html')


@cybersentry_ai.route('/ask', methods=['POST'])
def ask():
    try:
        question = request.json['question']
        print(f"Received question: {question}")


        answer, json_output = fuzzy_match(question, responses)
        print(f"JSON answer: {answer}")


        if answer:
            return jsonify({'answer': answer, 'source': 'JSON', 'terminal_output': json_output})
        else:
            print("No match found in JSON, trying Gemini API")
            gemini_answer, gemini_output = get_gemini_response(question)
            if gemini_answer:
                return jsonify({'answer': gemini_answer, 'source': 'Gemini', 'terminal_output':
gemini_output})
            else:
                fallback_answer = "Based on my current knowledge, I don't have a specific
answer to that question. However, in cybersecurity, it's important to always prioritize
data protection, use strong encryption, keep systems updated, and follow best
practices for network security."
                return jsonify({'answer': fallback_answer, 'source': 'Fallback', 'terminal_output':
''})
    except Exception as e:
        print(f"Error in /ask route: {e}")
        return jsonify({'error': str(e), 'terminal_output': ''}), 500


# This is important for Flask to recognize the blueprint
```

```
def init_app(app):
   app.register_blueprint(cybersentry_ai, url_prefix='/cybersenty_ai')
```

EXPLANATION:

I'll analyze your CyberSentry AI backend code in a well-structured format:

1) Packages - Usage

Core Packages:
- `flask`: Web framework for creating the API routes and blueprint
- `google.generativeai`: Google's Gemini API client for AI text generation
- `fuzzywuzzy`: Text matching library for finding similar questions
- `json`: Handles loading stored responses from JSON file
- `io.StringIO`: Captures stdout for logging and debugging purposes
- `sys`: Provides access to Python interpreter variables and functions

2) Classes and Functions - Usage and Explanation

Helper Functions:
- `load_responses()`:
  - Loads pre-defined question-answer pairs from a JSON file
  - Implements error handling for file loading issues
  - Returns an empty list as fallback if loading fails

- `capture_output(func)`:
  - Decorator that captures any print statements during function execution
  - Redirects stdout to a StringIO buffer temporarily
  - Returns both the function result and captured output
  - Useful for debugging and logging without modifying core functions

- `fuzzy_match(query, responses, threshold=80)`:
  - Performs fuzzy text matching to find similar questions in the knowledge base
  - Uses token set ratio algorithm with configurable match threshold (default 80%)
  - Returns the best matching answer if found, otherwise None
  - Captured by decorator to log matching process

- `get_gemini_response(query)`:
  - Connects to Google's Gemini API as fallback when local matching fails
  - Includes cybersecurity-specific context to guide the AI response
  - Handles API exceptions and returns None on failure
  - Captured by decorator to log API interaction details

3) Routes - Usage and Explanation

`@cybersentry_ai.route('/')`
- Renders the main cybersecurity AI assistant interface template
- Entry point for users to interact with the application

`@cybersentry_ai.route('/ask', methods=['POST'])`
- Main endpoint that handles user questions
- Implements a cascading answer strategy:
  1. First attempts to match question against local knowledge base
  2. Falls back to Gemini API if no local match is found
  3. Provides a generic cybersecurity answer as final fallback
- Returns JSON response with answer content, source identifier, and debug output
- Includes comprehensive error handling

4) Other Details

Blueprint Architecture:

- Uses Flask's Blueprint pattern for modular application design
- Enables the cybersecurity assistant to be mounted at a specific URL prefix
- Includes initialization function for Flask application registration

API Configuration:

- Configures Google Gemini API with hardcoded API key
- Uses 'gemini-1.5-flash-latest' model for optimal response speed
- Constructs context-aware prompts to generate security-focused responses

Error Handling:

- Multiple layers of try/except blocks to catch various failure scenarios
- Detailed error logging for troubleshooting
- Graceful degradation from local knowledge → API → generic fallback

Debugging Features:

- Custom decorator to capture stdout for debugging
- Terminal output included in API responses for transparency
- Detailed logging throughout the request handling process

5) Recap of the Code

This is a Flask-based cybersecurity assistant API that:

1. Implements a tiered response strategy combining local knowledge and AI
2. Uses fuzzy matching to find similar questions in a pre-defined database
3. Falls back to Google's Gemini AI model when local knowledge is insufficient
4. Provides detailed debugging information in the response
5. Follows modular design principles with Flask blueprints

The design prioritizes answering from known, vetted responses first before relying on the generative AI model. This approach balances accuracy (pre-defined answers) with flexibility (AI-generated responses).

Key improvements that could be considered:

1. Move the API key to environment variables for security
2. Implement caching for common queries
3. Add rate limiting to protect the API endpoint
4. Periodically update the local knowledge base with new Q&A pairs
5. Implement more sophisticated preprocessing of user questions

The code effectively demonstrates a hybrid approach to building an AI assistant that combines rule-based and machine learning techniques.

8) TRACKYLST:

CODE:

```
from flask import Blueprint, request, render_template, jsonify
import json
from collections import OrderedDict

osint = Blueprint('osint', __name__, template_folder='templates')

# Load JSON data
with open('data.json', 'r') as f:
    user_data = json.load(f)

# Define popular platforms to be displayed first
popular_platforms = ['Instagram', 'LinkedIn', 'Facebook', 'GitHub', 'LeetCode', 'Apple']
```

```
@osint.route('/')
def index():
    return render_template('osint.html')

@osint.route('/search', methods=['POST'])
def search():
    username = request.form['username']
    results = OrderedDict()

    # First, check popular platforms
    for platform in popular_platforms:
        if platform.lower() in user_data and 'url' in user_data[platform.lower()]:
            try:
                url = user_data[platform.lower()]['url'].format(username)
                results[platform] = url
            except KeyError:
                continue

    # Then, check all other platforms
    for site, site_data in sorted(user_data.items()):
        if site.capitalize() not in results and 'url' in site_data:
            try:
                url = site_data['url'].format(username)
                results[site.capitalize()] = url
            except KeyError:
                continue

    if results:
```

```
        return jsonify(results)
    else:
        return jsonify({"error": "No profiles found"}), 404
```

EXPLANATION:

I'll provide a comprehensive explanation of this Flask-based OSINT (Open Source Intelligence) web application code.

1) PACKAGES AND THEIR USAGE

- `flask`: Web framework providing core functionality
- `Blueprint`: For modular application structure
- `request`: Handles HTTP requests and form data
- `render_template`: Renders HTML templates
- `jsonify`: Converts Python dictionaries to JSON responses
- `json`: For reading and parsing JSON data
- `collections.OrderedDict`: Maintains ordered dictionary for consistent platform ordering

2) CLASS AND FUNCTION DEFINITIONS

Blueprint Definition:

```python
osint = Blueprint('osint', __name__, template_folder='templates')
```

- Creates a Flask Blueprint named 'osint'
- Sets up template directory for HTML files
- Enables modular organization of routes and views

JSON Loading:

```python
with open('data.json', 'r') as f:
    user_data = json.load(f)
```

- Reads platform configuration from data.json
- Stores platform URLs and related data

Popular Platforms List:

```python
popular_platforms = ['Instagram', 'LinkedIn', 'Facebook', 'GitHub', 'LeetCode', 'Apple']
```

- Defines priority platforms to be displayed first in search results

3) ROUTES AND THEIR USAGE

Index Route:

```python
@osint.route('/')
def index():
    return render_template('osint.html')
```

- Handles root URL requests
- Renders the main search interface (osint.html)

Search Route:

```python
@osint.route('/search', methods=['POST'])
def search():
```

- Handles POST requests to /search

- Processes username searches across platforms

- Returns JSON response with found profiles

4) OTHER IMPORTANT DETAILS

Search Logic:

- Two-phase search implementation:

 1. Checks popular platforms first:

 ```python

 for platform in popular_platforms:

    if platform.lower() in user_data and 'url' in user_data[platform.lower()]:

 ```

 2. Then searches remaining platforms:

 ```python

 for site, site_data in sorted(user_data.items()):

    if site.capitalize() not in results and 'url' in site_data:

 ```

Error Handling:

- Uses try-except blocks to handle missing username parameters

- Returns 404 status code if no profiles are found

- Uses KeyError exception handling for malformed URLs

URL Formatting:

- Uses string formatting to insert usernames into platform URLs

- Maintains consistent capitalization across platforms

5) CODE RECAP

This is a Flask-based web application for OSINT username searches:

- Implements a modular design using Flask Blueprint
- Loads platform configurations from an external JSON file
- Provides two endpoints: main page and search functionality
- Prioritizes popular social platforms in search results
- Returns formatted JSON responses with found profile URLs
- Includes basic error handling and status codes
- Uses ordered dictionaries to maintain consistent result ordering

The application follows a clean separation of concerns:

- Configuration data stored externally
- Modular routing structure
- Clear distinction between popular and other platforms
- Consistent error handling and response formatting

9) SITEINDEX:

CODE:

```
from flask import Flask, request, jsonify, render_template, Blueprint
import re
import dns.resolver
import requests
from urllib.parse import urlparse
import tldextract
import concurrent.futures
import whois
```

```
from datetime import datetime
import socket
import ssl
import OpenSSL
from bs4 import BeautifulSoup
from typing import Dict, List, Union

enscan = Blueprint('enscan', __name__, template_folder='templates')

@enscan.route('/')
def index():
   return render_template('enscan.html')

def is_valid_domain(domain: str) -> bool:
   pattern = r'^(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,}$'
   return bool(re.match(pattern, domain))

def get_ssl_info(domain: str) -> Dict:
   try:
      context = ssl.create_default_context()
      with socket.create_connection((domain, 443)) as sock:
         with context.wrap_socket(sock, server_hostname=domain) as ssock:
            cert = ssock.getpeercert()
            return {
               "issuer": dict(x[0] for x in cert['issuer']),
               "subject": dict(x[0] for x in cert['subject']),
               "version": cert['version'],
               "expires": cert['notAfter']
            }
```

```
    except Exception as e:
        return {"error": str(e)}


def get_security_headers(domain: str) -> Dict:
    try:
        response = requests.get(f"https://{domain}", timeout=5)
        headers = response.headers
        return {
            "X-Frame-Options": headers.get("X-Frame-Options", "Not set"),
            "X-XSS-Protection": headers.get("X-XSS-Protection", "Not set"),
            "Content-Security-Policy": headers.get("Content-Security-Policy", "Not set"),
            "Strict-Transport-Security": headers.get("Strict-Transport-Security", "Not set")
        }
    except Exception as e:
        return {"error": str(e)}


def get_domain_info(domain: str) -> Dict:
    try:
        w = whois.whois(domain)
        return {
            "registrar": w.registrar,
            "creation_date": w.creation_date.strftime("%Y-%m-%d") if
isinstance(w.creation_date, datetime) else str(w.creation_date),
            "expiration_date": w.expiration_date.strftime("%Y-%m-%d") if
isinstance(w.expiration_date, datetime) else str(w.expiration_date),
            "name_servers": w.name_servers if hasattr(w, 'name_servers') else None,
            "status": w.status if hasattr(w, 'status') else None,
            "emails": w.emails if hasattr(w, 'emails') else None
        }
    except Exception as e:
```

```
        return {"error": str(e)}


def check_spf_dmarc(domain: str) -> Dict:
    spf = dns_query(domain, 'TXT')
    dmarc = dns_query(f"_dmarc.{domain}", 'TXT')

    return {
        "spf_record": next((r for r in spf if "v=spf1" in str(r)), "No SPF record found"),
        "dmarc_record": next((r for r in dmarc if "v=DMARC1" in str(r)), "No DMARC record found")
    }

def dns_query(domain: str, record_type: str) -> Union[List[str], str]:
    resolver = dns.resolver.Resolver(configure=False)
    resolver.nameservers = ['8.8.8.8', '8.8.4.4']
    resolver.timeout = 5
    resolver.lifetime = 5

    try:
        answers = resolver.resolve(domain, record_type)
        return [str(record) for record in answers]
    except Exception as e:
        return str(e)

def dns_enumeration(domain: str) -> Dict:
    if not is_valid_domain(domain):
        return {"error": "Invalid domain format"}

    results = {}
    record_types = ['A', 'AAAA', 'NS', 'MX', 'TXT', 'SOA', 'CNAME', 'PTR', 'SRV']
```

```
    with concurrent.futures.ThreadPoolExecutor(max_workers=len(record_types)) as executor:
        future_to_record = {executor.submit(dns_query, domain, record_type): record_type
                            for record_type in record_types}
        for future in concurrent.futures.as_completed(future_to_record):
            record_type = future_to_record[future]
            results[record_type] = future.result()

    results.update({
        "domain_info": get_domain_info(domain),
        "ssl_info": get_ssl_info(domain),
        "security_headers": get_security_headers(domain),
        "email_security": check_spf_dmarc(domain)
    })

    return results

@enscan.route('/api/scan', methods=['POST'])
def scan():
    try:
        data = request.json
        input_value = data.get('input', '').strip()

        if not input_value:
            return jsonify({"error": "Empty input provided"}), 400

        result = dns_enumeration(input_value)
        return jsonify(result)
    except Exception as e:
```

```
        return jsonify({"error": str(e)}), 500


def create_app():
    app = Flask(__name__)
    app.register_blueprint(enscan)
    return app


if __name__ == '__main__':
    socket.setdefaulttimeout(10)
    app = create_app()
    app.run(debug=True, host='0.0.0.0', port=5000)
```

EXPLANATION:
I'll provide a detailed analysis of this DNS enumeration and security scanning Flask application.

1) PACKAGES AND THEIR USAGE

- `flask`: Web framework components
  - `Flask`: Core application object
  - `Blueprint`: Modular application structure
  - Other flask utilities: `request`, `jsonify`, `render_template`
- DNS and Network Related:
  - `dns.resolver`: DNS query handling
  - `socket`: Network connections
  - `ssl` & `OpenSSL`: SSL/TLS certificate handling
- Web and Parsing:
  - `requests`: HTTP requests
  - `BeautifulSoup`: HTML parsing
  - `urllib.parse`, `tldextract`: URL parsing
- Utility:

- `concurrent.futures`: Parallel processing
- `re`: Regular expressions
- `typing`: Type hints
- `whois`: Domain registration info
- `datetime`: Date handling

2) FUNCTIONS AND THEIR USAGE

Core Domain Validation:

```python
def is_valid_domain(domain: str) -> bool:
    pattern = r'^(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,}$'
    return bool(re.match(pattern, domain))
```

- Validates domain format using regex
- Returns boolean indicating validity

SSL Certificate Information:

```python
def get_ssl_info(domain: str) -> Dict:
```

- Establishes SSL connection
- Retrieves certificate details: issuer, subject, version, expiration

Security Headers Check:

```python
def get_security_headers(domain: str) -> Dict:
```

- Checks HTTP security headers

- Monitors X-Frame-Options, XSS Protection, CSP, HSTS

Domain Information:

```python
def get_domain_info(domain: str) -> Dict:
```

- Retrieves WHOIS information
- Returns registrar, dates, nameservers, status

Email Security:

```python
def check_spf_dmarc(domain: str) -> Dict:
```

- Checks SPF and DMARC records
- Verifies email security configurations

DNS Query Handler:

```python
def dns_query(domain: str, record_type: str) -> Union[List[str], str]:
```

- Performs DNS queries for specified record types
- Uses Google DNS servers (8.8.8.8, 8.8.4.4)

Main Enumeration Function:

```python
def dns_enumeration(domain: str) -> Dict:
```

- Orchestrates all scanning operations
- Uses concurrent execution for DNS queries

3) ROUTES AND THEIR USAGE

Index Route:

```python
@enscan.route('/')
def index():
   return render_template('enscan.html')
```

- Serves the main scanning interface

Scan API Route:

```python
@enscan.route('/api/scan', methods=['POST'])
def scan():
```

- Handles scan requests
- Processes domain input and returns results

4) OTHER IMPORTANT DETAILS

Error Handling:

- Comprehensive try-except blocks
- Detailed error messages
- Timeout configurations

Concurrency:

- ThreadPoolExecutor for parallel DNS queries
- Optimized performance for multiple record types

Security Features:

- Input validation

- Timeout limitations

- Secure default configurations

Application Configuration:

```python
def create_app():
   app = Flask(__name__)
   app.register_blueprint(enscan)
   return app
```

- Factory pattern for app creation

- Blueprint registration

5) CODE RECAP

This is a comprehensive domain scanning application that:

- Performs extensive DNS enumeration

- Checks security configurations

- Validates SSL certificates

- Verifies email security settings

- Uses concurrent processing for efficiency

- Provides detailed domain information

The application is well-structured with:

- Clear separation of concerns

- Type hints for better code clarity

- Comprehensive error handling
- Efficient resource usage
- Modular design using Blueprint
- Robust security checks

10) TRUESHOT _ AI:

CODE:

```
from flask import Flask, Blueprint, render_template, request, jsonify
from PIL import Image, ImageStat
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.models as models
import io
import numpy as np

trueshot_ai = Blueprint('trueshot_ai', __name__, url_prefix='/trueshot_ai')

class ImageClassifier:
    def __init__(self):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.model = self._setup_model()
        self.classes = ['AI-generated', 'Real']
        self.preprocess = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

```
        ])

    def _setup_model(self):
        model = models.resnet18(weights=None)
        num_ftrs = model.fc.in_features
        model.fc = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(num_ftrs, 2)
        )
        model.load_state_dict(torch.load('best_model9.pth', map_location=self.device))
        model.eval()
        return model.to(self.device)

    def _analyze_image_properties(self, image):
        # Convert to RGB if needed
        if image.mode in ('RGBA', 'P'):
            image = image.convert('RGB')

        # Basic properties
        width, height = image.size
        gray_image = image.convert('L')
        noise_level = float(np.std(np.array(gray_image)))

        # Advanced analysis
        stat = ImageStat.Stat(image)
        mean_colors = stat.mean
        std_colors = stat.stddev

        # Color variance analysis
```

```
        color_variance = sum(std_colors) / 3

        return {
            'dimensions': (width, height),
            'noise_level': noise_level,
            'color_variance': color_variance,
            'mean_brightness': sum(mean_colors) / 3
        }

    def _get_detailed_reasoning(self, prediction, confidence, properties):
        reasons = []
        noise = properties['noise_level']
        color_var = properties['color_variance']
        brightness = properties['mean_brightness']

        if confidence < 0.65:
            reasons.append("Analysis is inconclusive due to mixed characteristics")
            reasons.append(f"Confidence level ({confidence*100:.1f}%) is below threshold for definitive classification")
            return reasons

        if prediction == 'AI-generated':
            if noise < 15:
                reasons.append("Unusually smooth textures and low noise patterns typical of AI generation")
            if color_var < 30:
                reasons.append("Highly consistent color distributions across the image")
            if confidence > 0.85:
                reasons.append("Strong indicators of AI generation patterns")
```

```
            reasons.append(f"Image noise level ({noise:.1f}) is lower than typical natural photos")

        else:  # Real
            if noise > 15:
                reasons.append("Natural noise patterns consistent with real photography")
            if color_var > 30:
                reasons.append("Natural variation in color distribution")
            if confidence > 0.85:
                reasons.append("Strong indicators of natural photo characteristics")
            reasons.append(f"Image noise level ({noise:.1f}) matches typical camera sensor patterns")

        return reasons

    def classify_image(self, image):
        try:
            # Analyze image properties
            properties = self._analyze_image_properties(image)

            # Prepare image for model
            input_tensor = self.preprocess(image).unsqueeze(0).to(self.device)

            # Get model prediction
            with torch.no_grad():
                output = self.model(input_tensor)
                probabilities = torch.nn.functional.softmax(output[0], dim=0)
                confidence, predicted = torch.max(probabilities, 0)

            confidence_value = float(confidence.item())
```

```
            prediction = self.classes[predicted]

            # Get detailed reasoning
            reasoning = self._get_detailed_reasoning(prediction, confidence_value,
properties)

            return {
                'prediction': prediction,
                'confidence': confidence_value,
                'reasoning': reasoning
            }

        except Exception as e:
            raise Exception(f"Error during image classification: {str(e)}")

@trueshot_ai.route('/')
def index():
    return render_template('trueshot.html')

@trueshot_ai.route('/analyze', methods=['POST'])
def analyze():
    try:
        if 'image' not in request.files:
            return jsonify({
                'status': 'error',
                'message': 'No image file provided'
            }), 400

        file = request.files['image']
```

```
if file.filename == '':
    return jsonify({
        'status': 'error',
        'message': 'No selected file'
    }), 400

if not file.filename.lower().endswith(('.png', '.jpg', '.jpeg')):
    return jsonify({
        'status': 'error',
        'message': 'Invalid file type. Please upload JPG, JPEG or PNG'
    }), 400

# Read and validate image
image_bytes = file.read()
if not image_bytes:
    return jsonify({
        'status': 'error',
        'message': 'Empty file uploaded'
    }), 400

# Open and process image
image = Image.open(io.BytesIO(image_bytes))

# Convert to RGB if needed
if image.mode in ('RGBA', 'P'):
    image = image.convert('RGB')

# Get classification results
classifier_instance = ImageClassifier()
```

```
        result = classifier_instance.classify_image(image)

        return jsonify({
            'status': 'success',
            'result': result,
            'message': 'Analysis completed successfully'
        })

    except Exception as e:
        return jsonify({
            'status': 'error',
            'message': str(e)
        }), 500

if __name__ == '__main__':
    app = Flask(__name__)
    app.register_blueprint(trueshot_ai)
    app.run(debug=True, port=5000)
```

EXPLANATION:

I'll provide a detailed analysis of this AI Image Classification Flask application.

1) PACKAGES AND THEIR USAGE

- `flask`: Web framework components
  - `Flask`: Core application
  - `Blueprint`: Modular routing
  - Other utilities: `request`, `jsonify`, `render_template`
- Image Processing:
  - `PIL (Pillow)`: Image handling and analysis

- `ImageStat`: Image statistics calculation
- Deep Learning:
- `torch`: PyTorch deep learning framework
- `torchvision`: Computer vision utilities
- `models`: Pre-trained neural networks
- `transforms`: Image preprocessing
- Utility:
- `io`: Binary I/O operations
- `numpy`: Numerical computations

2) CLASSES AND FUNCTIONS

ImageClassifier Class:

```python
class ImageClassifier:
    def __init__(self):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.model = self._setup_model()
        self.classes = ['AI-generated', 'Real']
```

Key Methods:

- `_setup_model()`: Initializes ResNet18 model with custom classification head
- `_analyze_image_properties()`: Extracts image statistics
- `_get_detailed_reasoning()`: Provides analysis explanation
- `classify_image()`: Main classification method

Model Setup:

```python
def _setup_model(self):
```

```
    model = models.resnet18(weights=None)
    num_ftrs = model.fc.in_features
    model.fc = nn.Sequential(
        nn.Dropout(p=0.5),
        nn.Linear(num_ftrs, 2)
    )
```

- Uses ResNet18 architecture
- Custom classification layer
- Dropout for regularization

Image Analysis:

```python
def _analyze_image_properties(self, image):
    # Image property extraction
    width, height = image.size
    gray_image = image.convert('L')
    noise_level = float(np.std(np.array(gray_image)))
```

- Calculates dimensions
- Measures noise levels
- Analyzes color statistics

3) ROUTES AND THEIR USAGE

Index Route:

```python
@trueshot_ai.route('/')
def index():
```

```
    return render_template('trueshot.html')
```

- Serves main interface

Analysis Route:

```python
@trueshot_ai.route('/analyze', methods=['POST'])
def analyze():
```

- Handles image uploads
- Performs classification
- Returns detailed results

4) OTHER IMPORTANT DETAILS

Image Preprocessing:

```python
self.preprocess = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

- Resizes images
- Normalizes pixel values
- Prepares for model input

Error Handling:

- Comprehensive file validation
- Format checking
- Empty file detection
- Exception handling

Analysis Logic:
- Confidence thresholds
- Multiple analysis factors:
  - Noise patterns
  - Color distribution
  - Texture analysis
  - Brightness levels

5) CODE RECAP

This is a sophisticated AI image analysis application that:
- Detects AI-generated vs real images
- Uses deep learning (ResNet18)
- Provides detailed analysis reasoning
- Handles multiple image formats
- Performs comprehensive image property analysis

The application features:
- Modular design with Blueprint
- Robust error handling
- Detailed analysis reporting
- GPU acceleration when available
- Comprehensive image preprocessing
- Multiple analysis metrics

Technical Highlights:

- Custom neural network architecture
- Advanced image property analysis
- Confidence-based reasoning
- Efficient image processing pipeline
- Detailed result explanation system

11) WEBSEEKER:

CODE:

```
from flask import Flask, request, jsonify, render_template, Blueprint
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from flask_cors import CORS
import requests
import nmap
import re
import dns.resolver
import time
import logging
import socket
import whois
import ssl
import OpenSSL
import concurrent.futures
from datetime import datetime
from functools import lru_cache
```

```
from urllib.parse import urlparse
import os
import json

webseeker = Blueprint('webseeker', __name__, url_prefix='/webseeker')

# Configuration
class Config:
    VIRUSTOTAL_API_KEY = "4b4ae68cf38ed487342818091ad6ea879d11207e57049616f55fcd5c869233f9"
    IPINFO_API_KEY = "5425b721d94c6a"
    ABUSEIPDB_API_KEY = "040d7fe21b7908faa951fa8f5d9bbfe1e97cbae2294681d69626a0f4b681dfd9063b498948e02c7a"
    CACHE_DURATION = 3600  # 1 hour
    SCAN_TIMEOUT = 300  # 5 minutes
    MAX_CONCURRENT_SCANS = 3
    ALLOWED_SCAN_TYPES = ['quick', 'comprehensive', 'stealth']
    PORT_RANGES = {
        'quick': '20-25,53,80,110,143,443,465,587,993,995,3306,3389,5432,8080,8443',
        'comprehensive': '1-1024',
        'stealth': '21-23,25,53,80,110,111,135,139,143,443,445,993,995,1723,3306,3389,5900,8080'
    }

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler()
    ]
```

```
)

# Rate limiting configuration
limiter = Limiter(
    key_func=get_remote_address,
    default_limits=["100 per day", "10 per minute"],
    storage_uri="memory://"
)

class SecurityScanner:
    def __init__(self):
        self.nmap = nmap.PortScanner()
        self.scan_pool = concurrent.futures.ThreadPoolExecutor(
            max_workers=Config.MAX_CONCURRENT_SCANS
        )

    def validate_target(self, target):
        """Validate and normalize target input"""
        try:
            target = re.sub(r'^https?://', '', target)
            target = target.split('/')[0]

            if not re.match(r'^[\w\-\.]+\.[a-zA-Z]{2,}$', target):
                return None, "Invalid domain format"

            try:
                socket.gethostbyname(target)
            except socket.gaierror:
                return None, "Domain cannot be resolved"
```

```
            return target, None

        except Exception as e:
            logger.error(f"Target validation error: {str(e)}")
            return None, f"Validation error: {str(e)}"

    def get_ip_info(self, target):
        """Fetch enhanced IP and geolocation information"""
        try:
            ip = socket.gethostbyname(target)

            # Basic IP info from ipinfo.io
            response = requests.get(
                f"https://ipinfo.io/{ip}/json?token={Config.IPINFO_API_KEY}",
                timeout=10
            )
            response.raise_for_status()
            data = response.json()

            # Enhanced reverse DNS lookup
            try:
                answers = dns.resolver.resolve(dns.reversename.from_address(ip), "PTR")
                reverse_dns = str(answers[0]) if answers else None
            except Exception:
                reverse_dns = socket.getfqdn(ip)

            data['reverse_dns'] = reverse_dns
```

```
        # Enhanced WHOIS information
        try:
            whois_data = whois.whois(target)
            data['whois'] = {
                'registrar': whois_data.registrar,
                'creation_date': str(whois_data.creation_date[0]) if
isinstance(whois_data.creation_date, list) else str(whois_data.creation_date),
                'expiration_date': str(whois_data.expiration_date[0]) if
isinstance(whois_data.expiration_date, list) else str(whois_data.expiration_date),
                'name_servers': whois_data.name_servers if
isinstance(whois_data.name_servers, list) else [whois_data.name_servers] if
whois_data.name_servers else [],
                'status': whois_data.status if isinstance(whois_data.status, list) else
[whois_data.status] if whois_data.status else [],
                'emails': whois_data.emails if isinstance(whois_data.emails, list) else
[whois_data.emails] if whois_data.emails else []
            }
        except Exception:
            data['whois'] = {
                'registrar': None,
                'creation_date': None,
                'expiration_date': None,
                'name_servers': [],
                'status': [],
                'emails': []
            }

        # Get ASN details
        try:
            asn_response = requests.get(f"https://ipapi.co/{ip}/json/", timeout=5)
            if asn_response.status_code == 200:
```

```
            asn_data = asn_response.json()
            data.update({
                'asn': asn_data.get('asn'),
                'network': asn_data.get('network'),
                'network_type': asn_data.get('network_type'),
                'success': True
            })
        except Exception:
            pass

        # Add abuse contact information
        try:
            abuse_response = requests.get(
                f"https://api.abuseipdb.com/api/v2/check",
                params={'ipAddress': ip, 'maxAgeInDays': 90},
                headers={'Key': Config.ABUSEIPDB_API_KEY},
                timeout=5
            )
            if abuse_response.status_code == 200:
                abuse_data = abuse_response.json().get('data', {})
                data['abuse'] = {
                    'total_reports': abuse_data.get('totalReports'),
                    'confidence_score': abuse_data.get('abuseConfidenceScore'),
                    'last_reported': abuse_data.get('lastReportedAt'),
                    'is_public': abuse_data.get('isPublic'),
                    'abuse_types': abuse_data.get('reports', [])
                }
        except Exception:
            data['abuse'] = None
```

```
            return data

        except Exception as e:
            logger.error(f"IP info error for {target}: {str(e)}")
            return {"error": str(e)}

    def check_ssl(self, target):
        """Check SSL/TLS certificate information"""
        try:
            context = ssl.create_default_context()
            with socket.create_connection((target, 443)) as sock:
                with context.wrap_socket(sock, server_hostname=target) as ssock:
                    cert = ssock.getpeercert(True)
                    x509 = OpenSSL.crypto.load_certificate(
                        OpenSSL.crypto.FILETYPE_ASN1, cert
                    )

            issuer = {}
            for key, value in x509.get_issuer().get_components():
                issuer[key.decode('utf-8')] = value.decode('utf-8')

            subject = {}
            for key, value in x509.get_subject().get_components():
                subject[key.decode('utf-8')] = value.decode('utf-8')

            return {
                'issuer': issuer,
                'subject': subject,
```

```
            'version': x509.get_version(),
            'serial_number': str(x509.get_serial_number()),
            'not_before': x509.get_notBefore().decode('utf-8'),
            'not_after': x509.get_notAfter().decode('utf-8'),
            'expired': x509.has_expired(),
            'cipher': ssock.cipher(),
            'protocol': ssock.version()
        }
    except Exception as e:
        logger.error(f"SSL check error for {target}: {str(e)}")
        return {"error": str(e)}

def scan_virustotal(self, target):
    """Query VirusTotal API"""
    try:
        headers = {
            "x-apikey": Config.VIRUSTOTAL_API_KEY,
            "accept": "application/json"
        }

        # Get domain report
        domain_response = requests.get(
            f"https://www.virustotal.com/api/v3/domains/{target}",
            headers=headers,
            timeout=15
        )
        domain_response.raise_for_status()
        domain_data = domain_response.json()
```

```
    # Get URL report
    url_response = requests.get(
        f"https://www.virustotal.com/api/v3/urls/{target}",
        headers=headers,
        timeout=15
    )
    url_data = url_response.json() if url_response.status_code == 200 else {}

    return {
        'domain_report': domain_data.get('data', {}).get('attributes', {}),
        'url_report': url_data.get('data', {}).get('attributes', {}),
        'success': True
    }
except Exception as e:
    logger.error(f"VirusTotal error for {target}: {str(e)}")
    return {
        "data": {
            "attributes": {
                "last_analysis_stats": {
                    "harmless": 0,
                    "malicious": 0,
                    "suspicious": 0,
                    "unrated": 0
                }
            }
        },
        "success": False,
        "error": str(e)
    }
```

```
    def perform_port_scan(self, target, scan_type='quick'):
        """Perform Nmap port scan"""
        try:
            port_range = Config.PORT_RANGES.get(scan_type,
Config.PORT_RANGES['quick'])

            scan_args = {
                'quick': '-T4 -sV --version-intensity 5',
                'comprehensive': '-T4 -sV -sC -A --version-all',
                'stealth': '-T2 -sS -Pn --min-rate 100'
            }

            self.nmap.scan(target, port_range, arguments=scan_args[scan_type])

            processed_results = {
                'ports': [],
                'scan_stats': {
                    'start_time': datetime.now().isoformat(),
                    'elapsed': self.nmap.scanstats().get('elapsed', '0'),
                    'scan_type': scan_type
                },
                'os_detection': {},
                'success': True
            }

            if len(self.nmap.all_hosts()) > 0:
                host = self.nmap.all_hosts()[0]

                # Add OS detection if available
```

```
            if hasattr(self.nmap[host], 'osclass'):
                processed_results['os_detection'] = self.nmap[host]['osclass']

            for proto in self.nmap[host].all_protocols():
                ports = self.nmap[host][proto].keys()
                for port in ports:
                    port_info = self.nmap[host][proto][port]
                    processed_results['ports'].append({
                        'port': str(port),
                        'protocol': proto,
                        'state': port_info.get('state', ''),
                        'service': port_info.get('name', ''),
                        'version': port_info.get('version', ''),
                        'product': port_info.get('product', ''),
                        'extrainfo': port_info.get('extrainfo', ''),
                        'cpe': port_info.get('cpe', [])
                    })

            return processed_results

        except Exception as e:
            logger.error(f"Port scan error for {target}: {str(e)}")
            return {"error": str(e), "success": False}

    def perform_full_scan(self, target, scan_type):
        """Perform all security checks"""
        try:
            with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
                futures = {
```

```
            'ip_info': executor.submit(self.get_ip_info, target),
            'virustotal': executor.submit(self.scan_virustotal, target),
            'port_scan': executor.submit(self.perform_port_scan, target, scan_type),
            'ssl_cert': executor.submit(self.check_ssl, target)
        }

        results = {
            'timestamp': datetime.now().isoformat(),
            'target': target,
            'scan_type': scan_type,
            'success': True
        }

        for key, future in futures.items():
            try:
                results[key] = future.result(timeout=Config.SCAN_TIMEOUT)
            except concurrent.futures.TimeoutError:
                results[key] = {"error": "Scan timed out", "success": False}
            except Exception as e:
                results[key] = {"error": str(e), "success": False}

        return results

    except Exception as e:
        logger.error(f"Full scan error for {target}: {str(e)}")
        return {"error": f"Scan failed: {str(e)}", "success": False}

# Initialize scanner
scanner = SecurityScanner()
```

```
@webseeker.route('/')
def index():
    return render_template("webseeker.html")

@webseeker.route('/api/analyze', methods=['POST'])
@limiter.limit("10 per minute")
def analyze():
    try:
        data = request.get_json()
        target = data.get('domain', '').strip()
        scan_type = data.get('scanType', 'quick')

        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        if scan_type not in Config.ALLOWED_SCAN_TYPES:
            return jsonify({"error": "Invalid scan type", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        results = scanner.perform_full_scan(normalized_target, scan_type)

        if not results.get("success", False):
            return jsonify({"error": results.get("error", "Unknown error"), "success": False}),
500

        return jsonify(results)
```

```
    except Exception as e:
        logger.error(f"API error: {str(e)}")
        return jsonify({"error": "Internal server error", "success": False}), 500

@webseeker.route('/api/quick-check', methods=['GET'])
@limiter.limit("30 per minute")
def quick_check():
    """Lightweight domain check endpoint"""
    try:
        target = request.args.get('domain', '').strip()
        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        # Quick IP and VirusTotal check only
        with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
            ip_future = executor.submit(scanner.get_ip_info, normalized_target)
            vt_future = executor.submit(scanner.scan_virustotal, normalized_target)

            results = {
                'timestamp': datetime.now().isoformat(),
                'target': normalized_target,
                'ip_info': ip_future.result(timeout=10),
                'virustotal': vt_future.result(timeout=10),
                'success': True
```

```
        }

        return jsonify(results)

    except Exception as e:
        logger.error(f"Quick check error: {str(e)}")
        return jsonify({"error": str(e), "success": False}), 500

@webseeker.route('/api/ssl-check', methods=['GET'])
@limiter.limit("20 per minute")
def ssl_check():
    """Standalone SSL certificate checker"""
    try:
        target = request.args.get('domain', '').strip()
        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        ssl_info = scanner.check_ssl(normalized_target)
        ssl_info['success'] = 'error' not in ssl_info
        return jsonify(ssl_info)

    except Exception as e:
        logger.error(f"SSL check error: {str(e)}")
        return jsonify({"error": str(e), "success": False}), 500
```

```
@webseeker.route('/api/port-scan', methods=['POST'])
@limiter.limit("5 per minute")
def port_scan():
    """Standalone port scanner endpoint"""
    try:
        data = request.get_json()
        target = data.get('domain', '').strip()
        scan_type = data.get('scanType', 'quick')
        custom_ports = data.get('ports', None)

        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        # If custom ports provided, validate them
        if custom_ports:
            if not re.match(r'^[\d,\-]+$', custom_ports):
                return jsonify({"error": "Invalid port format", "success": False}), 400
            Config.PORT_RANGES['custom'] = custom_ports
            scan_type = 'custom'

        results = scanner.perform_port_scan(normalized_target, scan_type)
        return jsonify(results)

    except Exception as e:
        logger.error(f"Port scan error: {str(e)}")
```

```
        return jsonify({"error": str(e), "success": False}), 500


@webseeker.route('/api/abuse-check', methods=['GET'])
@limiter.limit("15 per minute")
def abuse_check():
    """Check IP address against AbuseIPDB"""
    try:
        target = request.args.get('domain', '').strip()
        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        ip = socket.gethostbyname(normalized_target)
        response = requests.get(
            "https://api.abuseipdb.com/api/v2/check",
            params={
                'ipAddress': ip,
                'maxAgeInDays': 90,
                'verbose': True
            },
            headers={'Key': Config.ABUSEIPDB_API_KEY},
            timeout=10
        )
        response.raise_for_status()

        return jsonify({
```

```
            'target': normalized_target,
            'ip': ip,
            'data': response.json().get('data', {}),
            'success': True
        })

    except Exception as e:
        logger.error(f"Abuse check error: {str(e)}")
        return jsonify({"error": str(e), "success": False}), 500

@webseeker.route('/api/whois', methods=['GET'])
@limiter.limit("15 per minute")
def whois_lookup():
    """Standalone WHOIS lookup endpoint"""
    try:
        target = request.args.get('domain', '').strip()
        if not target:
            return jsonify({"error": "No target specified", "success": False}), 400

        normalized_target, error = scanner.validate_target(target)
        if error:
            return jsonify({"error": error, "success": False}), 400

        whois_data = whois.whois(normalized_target)

        # Process and clean WHOIS data
        processed_data = {
            'domain': normalized_target,
            'registrar': whois_data.registrar,
```

```
        'creation_date': str(whois_data.creation_date[0]) if
isinstance(whois_data.creation_date, list) else str(whois_data.creation_date),
        'expiration_date': str(whois_data.expiration_date[0]) if
isinstance(whois_data.expiration_date, list) else str(whois_data.expiration_date),
        'name_servers': whois_data.name_servers if
isinstance(whois_data.name_servers, list) else [whois_data.name_servers] if
whois_data.name_servers else [],
        'status': whois_data.status if isinstance(whois_data.status, list) else
[whois_data.status] if whois_data.status else [],
        'emails': whois_data.emails if isinstance(whois_data.emails, list) else
[whois_data.emails] if whois_data.emails else [],
        'dnssec': whois_data.dnssec,
        'success': True
      }

      return jsonify(processed_data)

    except Exception as e:
      logger.error(f"WHOIS lookup error: {str(e)}")
      return jsonify({"error": str(e), "success": False}), 500

@webseeker.errorhandler(429)
def ratelimit_handler(e):
    return jsonify({"error": "Rate limit exceeded", "success": False}), 429

@webseeker.errorhandler(500)
def internal_error(e):
    logger.error(f"Internal server error: {str(e)}")
    return jsonify({"error": "Internal server error", "success": False}), 500

if __name__ == '__main__':
```

```
    logger.info("Starting security scanner server...")
    webseeker.run(debug=False, host='0.0.0.0', port=5000)
```

EXPLANATION:

I'll provide a comprehensive analysis of this web security scanning application.

1) PACKAGES AND THEIR USAGE

Core Framework:

- `flask`: Web application framework
- `flask_limiter`: Rate limiting
- `flask_cors`: Cross-origin resource sharing

Security and Network Tools:

- `nmap`: Port scanning
- `dns.resolver`: DNS lookups
- `whois`: Domain registration info
- `ssl`, `OpenSSL`: SSL/TLS certificate analysis
- `socket`: Network connections

Utility:

- `requests`: HTTP client
- `concurrent.futures`: Parallel processing
- `logging`: Application logging
- `datetime`, `time`: Time handling
- `json`, `re`: Data parsing

2) CLASSES AND FUNCTIONS

Configuration Class:

```python
class Config:
    VIRUSTOTAL_API_KEY = "..."
    IPINFO_API_KEY = "..."
    ABUSEIPDB_API_KEY = "..."
    SCAN_TIMEOUT = 300
    PORT_RANGES = {
        'quick': '20-25,53,80,110,143,443...',
        'comprehensive': '1-1024',
        'stealth': '21-23,25,53,80...'
    }
```

SecurityScanner Class:

Key Methods:

- `validate_target()`: Domain validation
- `get_ip_info()`: IP information gathering
- `check_ssl()`: SSL certificate analysis
- `scan_virustotal()`: VirusTotal API integration
- `perform_port_scan()`: Nmap scanning
- `perform_full_scan()`: Comprehensive scanning

3) ROUTES AND THEIR USAGE

Main Routes:

```python
@webseeker.route('/')
def index()
```

@webseeker.route('/api/analyze', methods=['POST'])

@limiter.limit("10 per minute")

def analyze()

```

Specialized Endpoints:

```python

@webseeker.route('/api/quick-check', methods=['GET'])

@webseeker.route('/api/ssl-check', methods=['GET'])

@webseeker.route('/api/port-scan', methods=['POST'])

@webseeker.route('/api/abuse-check', methods=['GET'])

@webseeker.route('/api/whois', methods=['GET'])

```

4) IMPORTANT FEATURES

Security Measures:

- Rate limiting:

```python

limiter = Limiter(

  key_func=get_remote_address,

  default_limits=["100 per day", "10 per minute"]

)

```

Concurrent Processing:

```python

with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:

```
    futures = {
        'ip_info': executor.submit(self.get_ip_info, target),
        'virustotal': executor.submit(self.scan_virustotal, target),
        'port_scan': executor.submit(self.perform_port_scan, target, scan_type),
        'ssl_cert': executor.submit(self.check_ssl, target)
    }
```

Error Handling:

- Comprehensive try-except blocks
- Custom error handlers
- Detailed logging

5) CODE RECAP

This is a sophisticated web security scanning application that:

- Performs comprehensive domain analysis
- Integrates multiple security APIs
- Handles concurrent scanning operations
- Implements rate limiting
- Provides detailed security reports

Key Features:

- Multiple scan types (quick, comprehensive, stealth)
- Port scanning with Nmap
- SSL certificate analysis
- WHOIS lookup
- IP abuse checking
- VirusTotal integration

- Concurrent execution
- Rate limiting
- Detailed error handling
- Logging system

Security Considerations:

- Input validation
- Rate limiting
- Timeout mechanisms
- Error handling
- API key protection

The application provides a robust security scanning platform with:

- Modular design
- Scalable architecture
- Comprehensive error handling
- Multiple API integrations
- Detailed reporting capabilities

TRUESHOT ML MODEL EXPLANATION:

The model you're working with appears to be a custom ResNet-based neural network built using PyTorch, tailored for binary classification. Here's a detailed breakdown of your model and its components:

---

1. Base Model

The foundation of your model is a pre-trained ResNet-18 (a popular deep learning model for image classification). ResNet-18 is part of the ResNet (Residual Network) family, which uses residual connections (or "skip connections") to improve gradient flow and mitigate the vanishing gradient problem.

Key details about ResNet-18:

Architecture: 18 layers of convolutional and fully connected layers, organized into blocks.

Pre-training: You are leveraging weights pre-trained on the ImageNet dataset, which allows your model to benefit from features learned on a large-scale image dataset.

Modification: You replace the final fully connected layer (fc) to adapt ResNet-18 to your specific binary classification task.

---

2. Custom ResNet Model

Your custom model (ResNetModel) builds on top of ResNet-18. Here's what happens step-by-step:

Model Definition

self.model = models.resnet18(pretrained=True)

Loads a pre-trained ResNet-18 model.

pretrained=True ensures that weights from ImageNet training are used.

Modifications

```
self.model.fc = nn.Sequential(
  nn.Linear(in_features, 256),
  nn.ReLU(),
  nn.Linear(256, num_classes)
)
```

self.model.fc: The default final fully connected layer is replaced with a custom head.

The input features (in_features) of this layer come from ResNet's global average pooling (GAP) layer output, typically 512.

The new fully connected layers:

1. A linear layer reducing 512 features to 256 features.

2. A ReLU activation function for non-linearity.

3. Another linear layer mapping 256 features to num_classes (set to 2 for binary classification).

Purpose of the Custom Layers

To adapt the generic ResNet-18 model to your specific binary classification task.

The added layers act as a "classifier head," learning task-specific features while preserving the general feature extraction capabilities of the pre-trained ResNet-18.

---

3. Dataset and Task

From your code and logs, the model is designed for a binary classification task. This likely involves:

Input Data: Images passed as tensors (3-channel RGB images of size 224x224, as required by ResNet-18).

Output: Probabilities for two classes, e.g., 0 (class 1) and 1 (class 2).

The output is generated using softmax to compute probabilities for each class:

probabilities = torch.softmax(output, dim=1)

Example:

For an input image, the model predicts:

Class 0: 50.09%

Class 1: 49.91%

---

4. Training Process

The model is typically trained as follows:

1. Dataset Preparation:

Images are preprocessed (resized, normalized).

Data is split into training and validation sets.

2. Optimization:

Loss Function: Likely Cross-Entropy Loss for binary classification.

Optimizer: Common choices include Adam or SGD.

3. Fine-Tuning:

The pre-trained ResNet-18 weights are fine-tuned on your dataset.

The earlier layers (feature extraction) are mostly frozen, while the new fc layers are trained from scratch.

---

5. Model Workflow

Here's how the model operates:

1. Input: An image tensor of size (3, 224, 224).

2. Feature Extraction: The input passes through the convolutional and residual blocks of ResNet-18 to extract high-level features.

3. Classification Head:

The extracted features are passed to the fully connected layers (self.model.fc).

The output is a tensor of logits, which is converted to probabilities using softmax.

4. Output: A tensor with probabilities for the two classes.

---

6. Model Advantages

1. Pre-trained Weights: Leverages ImageNet knowledge, reducing training time and improving performance on smaller datasets.

2. Residual Connections: Improves gradient flow and learning stability in deep networks.

3. Custom Layers: Makes the model flexible and tailored to your specific task.

---

7. Logs from Execution

Your logs demonstrate:

The model successfully completed a forward pass, producing valid logits.

The probabilities indicate the relative confidence for the two classes.

The predicted class (0 in this case) corresponds to the higher probability.

---

8. Applications

Your model can be used in various binary classification tasks such as:

Image authenticity detection (aligned with your project, Trueshot_AI).

Anomaly detection in images.

Object presence/absence detection.

Simple explanation:
Got it, Lakshan! I'll break everything down in a **very simple and detailed** way so you understand **everything** about your ResNet-18 model and how it works for classifying **real vs. AI-generated images**.

🔹 1. What is ResNet-18?

Imagine you have a **super-smart detective** 🕵️‍♂️ who can recognize whether a picture is real or AI-generated. This detective has to analyze every tiny detail—textures, edges, lighting, and patterns—to make a decision.

ResNet-18 is like that detective, but instead of using **human eyes**, it uses **math and deep learning** to look at images and decide if they are real or AI-generated.

💡 **ResNet-18 stands for "Residual Network-18"**

- **"Residual"** means it uses **shortcut connections** to learn better.
- **"18"** means it has **18 layers** that analyze different features of an image.
- It is a type of **Convolutional Neural Network (CNN)** used for **image classification**.

🔹 **Why use ResNet-18?**

✅ **Smaller model, faster training** (compared to bigger models like ResNet-50 or ResNet-101).

✅ **Performs well even with fewer images**.

✅ **Uses "skip connections"** to avoid problems like the **vanishing gradient** (I'll explain this later).

🔹 **2. How does ResNet-18 work? (Step by Step)**

Think of an image as a **complex puzzle** 🧩. To understand whether it's real or AI-generated, the model **breaks the image down into layers**, each layer learning something different.

Step 1: Image Processing

- ResNet-18 starts by looking at **small details** like **edges, colors, and textures**.
- It then **builds up** to recognize **bigger features** like shapes, faces, and objects.

Step 2: Convolution Layers

- The model has **18 layers**, and each layer detects **different parts of the image**.
- The first layers detect **simple patterns** (like lines, corners, and edges).
- The deeper layers detect **complex features** (like faces, text, or lighting patterns).

Step 3: Skip Connections (Shortcut Paths)

Normally, when a model has many layers, it can **forget important details** as information moves forward.

💡 **Imagine playing a long game of Chinese whispers**—the message gets lost as more people pass it along! 🤦

ResNet-18 **solves this problem using "skip connections"** 🚀

- It **skips some layers** and **sends the information directly forward**.
- This helps the model **remember important details** and prevents the "vanishing gradient problem" (where learning stops in deep networks).

🔹 3. How Your Model is Different from Base ResNet-18

A normal **ResNet-18 (Base Model)** is trained on a dataset called **ImageNet**, which contains **millions of general images** (cats, dogs, cars, airplanes, etc.).

Your model, however, is **custom-trained** for a **specific task**:
✅ **Classifying whether an image is real or AI-generated**.

What Changes Were Made to ResNet-18?

1️⃣ Replaced the Last Layer

- Normal ResNet-18 has **1000 output classes** (for general objects).
- Your model **only has 2 classes**: **"Real" and "AI-Generated"**.
- The final layer (called fc) was changed to:
- model.fc = nn.Linear(num_ftrs, 2) # 2 output classes

2️⃣ Trained on a Special Dataset

- Instead of training on general objects, your model was trained on **real vs. AI-generated images**.
- This helps it **specialize in detecting AI-generated content**.

3️⃣ Fine-Tuning

- Instead of training from scratch (which takes a long time), you **fine-tuned** the model by adjusting only the last few layers.
- This makes training **faster and more efficient**.

🔹 4. Accuracy Comparison (Base Model vs. Your Model)

Model	Dataset Used	Accuracy (Approx.)
Base ResNet-18	ImageNet (General Images)	~69% (Top-1)
Your Trained Model	Real vs. AI Images	**?% (Check below)**

💡 **How to Check Your Model's Accuracy?**
If you trained the model in PyTorch, you can check accuracy using:

model.eval() # Set model to evaluation mode

correct = 0

total = 0

```
with torch.no_grad():
    for images, labels in test_loader:  # test_loader contains test images
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Model Accuracy: {accuracy:.2f}%")
```

🔹 **5. How to Train ResNet-18 for Real vs. AI Image Classification**

Step 1: Load the Pre-Trained ResNet-18 Model

```
import torch
import torch.nn as nn
import torchvision.models as models

# Load ResNet-18 with pre-trained weights
model = models.resnet18(pretrained=True)

# Modify the last layer to classify 2 categories: Real & AI-generated
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)  # Output: 2 classes (0=Real, 1=AI)
```

Step 2: Prepare the Dataset

You need a dataset with **two folders**:

✅ **Folder 1:** "Real" images
✅ **Folder 2:** "AI-Generated" images

Example code to load images:

```
from torchvision import datasets, transforms
```

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to match ResNet input size
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
])

# Load dataset
train_dataset = datasets.ImageFolder(root='path_to_train_data', transform=transform)
test_dataset = datasets.ImageFolder(root='path_to_test_data', transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Step 3: Train the Model

```
import torch.optim as optim

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}")
```

Step 4: Save and Test the Model

```
# Save the trained model
torch.save(model.state_dict(), "resnet18_real_vs_ai.pth")

# Load and test the model
model.load_state_dict(torch.load("resnet18_real_vs_ai.pth"))
model.eval()

# Test on a new image
from PIL import Image

image = Image.open("test_image.jpg")
image = transform(image).unsqueeze(0)  # Add batch dimension

output = model(image)
_, predicted = torch.max(output, 1)

print("Prediction:", "Real" if predicted.item() == 0 else "AI-Generated")
```

🔹 **Summary**

✅ **ResNet-18 is a deep learning model** used for **image classification**.
✅ Your model **modifies** it to classify **Real vs. AI-generated images**.
✅ Training involves:

- **Using a pre-trained ResNet-18 model**
- **Fine-tuning it with custom data**
- **Evaluating accuracy**

Would you like help testing your trained model? 🚀