**1. Advanced Quantum Computing Integration**

Integrate quantum computing capabilities to perform complex calculations and simulations that are beyond the reach of classical computers.

python

Copy

# quantum\_computing.py

import threading

import time

from qiskit import QuantumCircuit, Aer, execute

class QuantumComputing:

def \_\_init\_\_(self, config):

self.config = config

self.simulator = Aer.get\_backend('qasm\_simulator')

self.lock = threading.Lock()

def create\_quantum\_circuit(self, num\_qubits):

with self.lock:

qc = QuantumCircuit(num\_qubits)

return qc

def add\_gate(self, qc, gate, qubits):

with self.lock:

if gate == 'h':

qc.h(qubits)

elif gate == 'x':

qc.x(qubits)

elif gate == 'y':

qc.y(qubits)

elif gate == 'z':

qc.z(qubits)

elif gate == 'cx':

qc.cx(qubits[0], qubits[1])

elif gate == 'swap':

qc.swap(qubits[0], qubits[1])

else:

raise ValueError(f"Unknown gate: {gate}")

return qc

def execute\_circuit(self, qc, shots=1024):

with self.lock:

job = execute(qc, self.simulator, shots=shots)

result = job.result()

counts = result.get\_counts(qc)

return counts

def continuous\_quantum\_computing(self):

def quantum\_loop():

while True:

time.sleep(3600) # Perform quantum computations every hour

# Implement continuous quantum computing logic

pass

threading.Thread(target=quantum\_loop, daemon=True).start()

**2. Advanced Blockchain Integration**

Integrate blockchain technology to enable secure and transparent transactions, smart contracts, and decentralized applications.

python

Copy

# blockchain\_integration.py

import threading

import time

from web3 import Web3

class BlockchainIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.web3 = Web3(Web3.HTTPProvider(config.get('blockchain\_provider', 'http://localhost:8545')))

self.lock = threading.Lock()

def get\_balance(self, address):

with self.lock:

balance = self.web3.eth.get\_balance(address)

return self.web3.from\_wei(balance, 'ether')

def send\_transaction(self, from\_address, to\_address, amount, private\_key):

with self.lock:

nonce = self.web3.eth.get\_transaction\_count(from\_address)

tx = {

'nonce': nonce,

'to': to\_address,

'value': self.web3.to\_wei(amount, 'ether'),

'gas': 2000000,

'gasPrice': self.web3.to\_wei('50', 'gwei')

}

signed\_tx = self.web3.eth.account.sign\_transaction(tx, private\_key)

tx\_hash = self.web3.eth.send\_raw\_transaction(signed\_tx.rawTransaction)

return tx\_hash.hex()

def deploy\_contract(self, contract\_interface, constructor\_args, private\_key):

with self.lock:

contract = self.web3.eth.contract(abi=contract\_interface['abi'], bytecode=contract\_interface['bytecode'])

tx = contract.constructor(\*constructor\_args).build\_transaction({

'from': self.web3.eth.account.from\_key(private\_key).address,

'nonce': self.web3.eth.get\_transaction\_count(self.web3.eth.account.from\_key(private\_key).address),

'gas': 2000000,

'gasPrice': self.web3.to\_wei('50', 'gwei')

})

signed\_tx = self.web3.eth.account.sign\_transaction(tx, private\_key)

tx\_hash = self.web3.eth.send\_raw\_transaction(signed\_tx.rawTransaction)

return tx\_hash.hex()

def continuous\_blockchain\_integration(self):

def blockchain\_loop():

while True:

time.sleep(3600) # Integrate with blockchain every hour

# Implement continuous blockchain integration logic

pass

threading.Thread(target=blockchain\_loop, daemon=True).start()

**3. Advanced Augmented Reality (AR) and Virtual Reality (VR) Integration**

Integrate AR and VR technologies to create immersive and interactive experiences for users.

python

Copy

# ar\_vr\_integration.py

import threading

import time

import pyautogui

import cv2

import numpy as np

class ARVRIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def start\_ar\_session(self):

with self.lock:

# Implement AR session logic

pass

def start\_vr\_session(self):

with self.lock:

# Implement VR session logic

pass

def overlay\_ar\_content(self, content):

with self.lock:

screenshot = pyautogui.screenshot()

screenshot = cv2.cvtColor(np.array(screenshot), cv2.COLOR\_RGB2BGR)

# Implement AR content overlay logic

cv2.imshow('AR Overlay', screenshot)

cv2.waitKey(0)

cv2.destroyAllWindows()

def continuous\_ar\_vr\_integration(self):

def ar\_vr\_loop():

while True:

time.sleep(3600) # Integrate with AR/VR every hour

# Implement continuous AR/VR integration logic

pass

threading.Thread(target=ar\_vr\_loop, daemon=True).start()

**4. Advanced Internet of Things (IoT) Integration**

Integrate IoT devices to enable remote monitoring, control, and automation of various systems and appliances.

python

Copy

# iot\_integration.py

import threading

import time

import paho.mqtt.client as mqtt

class IoTIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.mqtt\_client = mqtt.Client()

self.mqtt\_client.on\_connect = self.on\_connect

self.mqtt\_client.on\_message = self.on\_message

self.mqtt\_client.connect(config.get('mqtt\_broker', 'localhost'), config.get('mqtt\_port', 1883), 60)

self.lock = threading.Lock()

def on\_connect(self, client, userdata, flags, rc):

print(f"Connected with result code {rc}")

client.subscribe("ultron/iot/#")

def on\_message(self, client, userdata, msg):

print(f"Received message on topic {msg.topic}: {msg.payload.decode()}")

def publish\_message(self, topic, message):

with self.lock:

self.mqtt\_client.publish(topic, message)

def control\_device(self, device\_id, command):

with self.lock:

topic = f"ultron/iot/{device\_id}/control"

self.publish\_message(topic, command)

def monitor\_device(self, device\_id):

with self.lock:

topic = f"ultron/iot/{device\_id}/monitor"

self.mqtt\_client.subscribe(topic)

def continuous\_iot\_integration(self):

def iot\_loop():

while True:

time.sleep(3600) # Integrate with IoT every hour

# Implement continuous IoT integration logic

pass

threading.Thread(target=iot\_loop, daemon=True).start()

**5. Advanced Edge Computing Integration**

Integrate edge computing capabilities to enable real-time processing and analysis of data at the edge of the network.

python

Copy

# edge\_computing.py

import threading

import time

import edgeiq

class EdgeComputing:

def \_\_init\_\_(self, config):

self.config = config

self.fps = edgeiq.FPS()

self.lock = threading.Lock()

def start\_edge\_computing(self):

with self.lock:

# Implement edge computing logic

pass

def process\_frame(self, frame):

with self.lock:

# Implement frame processing logic

pass

def continuous\_edge\_computing(self):

def edge\_loop():

while True:

time.sleep(3600) # Perform edge computing every hour

# Implement continuous edge computing logic

pass

threading.Thread(target=edge\_loop, daemon=True).start()

**6. Advanced Federated Learning Integration**

Integrate federated learning capabilities to enable collaborative training of machine learning models across multiple devices without sharing raw data.

python

Copy

# federated\_learning.py

import threading

import time

import flwr as fl

class FederatedLearning:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def start\_federated\_learning(self):

with self.lock:

# Implement federated learning logic

pass

def train\_model(self, model, data):

with self.lock:

# Implement model training logic

pass

def aggregate\_models(self, models):

with self.lock:

# Implement model aggregation logic

pass

def continuous\_federated\_learning(self):

def fl\_loop():

while True:

time.sleep(3600) # Perform federated learning every hour

# Implement continuous federated learning logic

pass

threading.Thread(target=fl\_loop, daemon=True).start()

**7. Advanced Explainable AI (XAI) Integration**

Integrate XAI techniques to provide transparent and interpretable explanations for AI decisions and predictions.

python

Copy

# xai\_integration.py

import threading

import time

import shap

import lime

class XAIIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def explain\_prediction(self, model, data):

with self.lock:

# Implement prediction explanation logic

pass

def generate\_shap\_values(self, model, data):

with self.lock:

explainer = shap.Explainer(model)

shap\_values = explainer(data)

return shap\_values

def generate\_lime\_explanation(self, model, data):

with self.lock:

explainer = lime.LimeTabularExplainer(data, mode='regression')

explanation = explainer.explain\_instance(data[0], model.predict)

return explanation

def continuous\_xai\_integration(self):

def xai\_loop():

while True:

time.sleep(3600) # Perform XAI every hour

# Implement continuous XAI integration logic

pass

threading.Thread(target=xai\_loop, daemon=True).start()

**8. Advanced Autonomous Systems Integration**

Integrate autonomous systems capabilities to enable self-driving cars, drones, and other autonomous vehicles and devices.

python

Copy

# autonomous\_systems.py

import threading

import time

import airsim

class AutonomousSystems:

def \_\_init\_\_(self, config):

self.config = config

self.client = airsim.MultirotorClient()

self.client.confirmConnection()

self.client.enableApiControl(True)

self.lock = threading.Lock()

def takeoff(self):

with self.lock:

self.client.takeoffAsync().join()

def land(self):

with self.lock:

self.client.landAsync().join()

def move\_to\_position(self, x, y, z, velocity):

with self.lock:

self.client.moveToPositionAsync(x, y, z, velocity).join()

def take\_picture(self):

with self.lock:

responses = self.client.simGetImages([airsim.ImageRequest("0", airsim.ImageType.Scene, False, False)])

response = responses[0]

img1d = np.frombuffer(response.image\_data\_uint8, dtype=np.uint8)

img\_rgba = img1d.reshape(response.height, response.width, 4)

img\_rgba = cv2.cvtColor(img\_rgba, cv2.COLOR\_RGBA2RGB)

return img\_rgba

def continuous\_autonomous\_systems(self):

def autonomous\_loop():

while True:

time.sleep(3600) # Perform autonomous operations every hour

# Implement continuous autonomous systems logic

pass

threading.Thread(target=autonomous\_loop, daemon=True).start()

**9. Advanced Robotics Integration**

Integrate robotics capabilities to enable the control and automation of physical robots and machines.

python

Copy

# robotics\_integration.py

import threading

import time

import pyrobot

class RoboticsIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.bot = pyrobot.Robot(config.get('robot\_config', 'config.json'))

self.lock = threading.Lock()

def move\_forward(self, distance):

with self.lock:

self.bot.base.go\_forward(distance)

def move\_backward(self, distance):

with self.lock:

self.bot.base.go\_backward(distance)

def turn\_left(self, angle):

with self.lock:

self.bot.base.turn\_left(angle)

def turn\_right(self, angle):

with self.lock:

self.bot.base.turn\_right(angle)

def take\_picture(self):

with self.lock:

image = self.bot.camera.get\_image()

return image

def continuous\_robotics\_integration(self):

def robotics\_loop():

while True:

time.sleep(3600) # Perform robotics operations every hour

# Implement continuous robotics integration logic

pass

threading.Thread(target=robotics\_loop, daemon=True).start()

**10. Advanced Cybersecurity Integration**

Integrate cybersecurity capabilities to protect systems and data from cyber threats and attacks.

python

Copy

# cybersecurity\_integration.py

import threading

import time

import nmap

class CybersecurityIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.nm = nmap.PortScanner()

self.lock = threading.Lock()

def scan\_network(self, target):

with self.lock:

self.nm.scan(hosts=target, arguments='-sS -p 1-1000')

return self.nm.csv()

def detect\_vulnerabilities(self, target):

with self.lock:

# Implement vulnerability detection logic

pass

def secure\_system(self, target):

with self.lock:

# Implement system security logic

pass

def continuous\_cybersecurity\_integration(self):

def cybersecurity\_loop():

while True:

time.sleep(3600) # Perform cybersecurity operations every hour

# Implement continuous cybersecurity integration logic

pass

threading.Thread(target=cybersecurity\_loop, daemon=True).start()

**11. Advanced Data Privacy Integration**

Integrate data privacy capabilities to ensure the protection and secure handling of sensitive and personal data.

python

Copy

# data\_privacy.py

import threading

import time

from cryptography.fernet import Fernet

class DataPrivacy:

def \_\_init\_\_(self, config):

self.config = config

self.key = Fernet.generate\_key()

self.cipher\_suite = Fernet(self.key)

self.lock = threading.Lock()

def encrypt\_data(self, data):

with self.lock:

if isinstance(data, str):

data = data.encode()

encrypted\_data = self.cipher\_suite.encrypt(data)

return encrypted\_data

def decrypt\_data(self, encrypted\_data):

with self.lock:

decrypted\_data = self.cipher\_suite.decrypt(encrypted\_data)

return decrypted\_data.decode()

def anonymize\_data(self, data):

with self.lock:

# Implement data anonymization logic

pass

def continuous\_data\_privacy(self):

def privacy\_loop():

while True:

time.sleep(3600) # Perform data privacy operations every hour

# Implement continuous data privacy logic

pass

threading.Thread(target=privacy\_loop, daemon=True).start()

**12. Advanced Ethical AI Integration**

Integrate ethical AI capabilities to ensure that AI systems are designed and operated in a manner that is fair, transparent, and accountable.

python

Copy

# ethical\_ai.py

import threading

import time

import json

class EthicalAI:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_principles = self.\_load\_ethical\_principles()

self.lock = threading.Lock()

def \_load\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical principles: {e}")

return {}

def \_save\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'w') as f:

json.dump(self.ethical\_principles, f, indent=4)

except Exception as e:

print(f"Error saving ethical principles: {e}")

def evaluate\_action(self, action):

with self.lock:

for principle, value in self.ethical\_principles.items():

if not self.\_check\_principle(action, principle, value):

return False

return True

def \_check\_principle(self, action, principle, value):

# Implement principle checking logic

return True

def update\_principles(self, new\_principles):

with self.lock:

self.ethical\_principles.update(new\_principles)

self.\_save\_ethical\_principles()

def continuous\_ethical\_ai(self):

def ethical\_loop():

while True:

time.sleep(3600) # Perform ethical AI operations every hour

# Implement continuous ethical AI logic

pass

threading.Thread(target=ethical\_loop, daemon=True).start()

**13. Advanced Explainable AI (XAI) Integration**

Integrate XAI techniques to provide transparent and interpretable explanations for AI decisions and predictions.

python

Copy

# xai\_integration.py

import threading

import time

import shap

import lime

class XAIIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def explain\_prediction(self, model, data):

with self.lock:

# Implement prediction explanation logic

pass

def generate\_shap\_values(self, model, data):

with self.lock:

explainer = shap.Explainer(model)

shap\_values = explainer(data)

return shap\_values

def generate\_lime\_explanation(self, model, data):

with self.lock:

explainer = lime.LimeTabularExplainer(data, mode='regression')

explanation = explainer.explain\_instance(data[0], model.predict)

return explanation

def continuous\_xai\_integration(self):

def xai\_loop():

while True:

time.sleep(3600) # Perform XAI every hour

# Implement continuous XAI integration logic

pass

threading.Thread(target=xai\_loop, daemon=True).start()

**14. Advanced Autonomous Systems Integration**

Integrate autonomous systems capabilities to enable self-driving cars, drones, and other autonomous vehicles and devices.

python

Copy

# autonomous\_systems.py

import threading

import time

import airsim

class AutonomousSystems:

def \_\_init\_\_(self, config):

self.config = config

self.client = airsim.MultirotorClient()

self.client.confirmConnection()

self.client.enableApiControl(True)

self.lock = threading.Lock()

def takeoff(self):

with self.lock:

self.client.takeoffAsync().join()

def land(self):

with self.lock:

self.client.landAsync().join()

def move\_to\_position(self, x, y, z, velocity):

with self.lock:

self.client.moveToPositionAsync(x, y, z, velocity).join()

def take\_picture(self):

with self.lock:

responses = self.client.simGetImages([airsim.ImageRequest("0", airsim.ImageType.Scene, False, False)])

response = responses[0]

img1d = np.frombuffer(response.image\_data\_uint8, dtype=np.uint8)

img\_rgba = img1d.reshape(response.height, response.width, 4)

img\_rgba = cv2.cvtColor(img\_rgba, cv2.COLOR\_RGBA2RGB)

return img\_rgba

def continuous\_autonomous\_systems(self):

def autonomous\_loop():

while True:

time.sleep(3600) # Perform autonomous operations every hour

# Implement continuous autonomous systems logic

pass

threading.Thread(target=autonomous\_loop, daemon=True).start()

**15. Advanced Robotics Integration**

Integrate robotics capabilities to enable the control and automation of physical robots and machines.

python

Copy

# robotics\_integration.py

import threading

import time

import pyrobot

class RoboticsIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.bot = pyrobot.Robot(config.get('robot\_config', 'config.json'))

self.lock = threading.Lock()

def move\_forward(self, distance):

with self.lock:

self.bot.base.go\_forward(distance)

def move\_backward(self, distance):

with self.lock:

self.bot.base.go\_backward(distance)

def turn\_left(self, angle):

with self.lock:

self.bot.base.turn\_left(angle)

def turn\_right(self, angle):

with self.lock:

self.bot.base.turn\_right(angle)

def take\_picture(self):

with self.lock:

image = self.bot.camera.get\_image()

return image

def continuous\_robotics\_integration(self):

def robotics\_loop():

while True:

time.sleep(3600) # Perform robotics operations every hour

# Implement continuous robotics integration logic

pass

threading.Thread(target=robotics\_loop, daemon=True).start()

**16. Advanced Cybersecurity Integration**

Integrate cybersecurity capabilities to protect systems and data from cyber threats and attacks.

python

Copy

# cybersecurity\_integration.py

import threading

import time

import nmap

class CybersecurityIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.nm = nmap.PortScanner()

self.lock = threading.Lock()

def scan\_network(self, target):

with self.lock:

self.nm.scan(hosts=target, arguments='-sS -p 1-1000')

return self.nm.csv()

def detect\_vulnerabilities(self, target):

with self.lock:

# Implement vulnerability detection logic

pass

def secure\_system(self, target):

with self.lock:

# Implement system security logic

pass

def continuous\_cybersecurity\_integration(self):

def cybersecurity\_loop():

while True:

time.sleep(3600) # Perform cybersecurity operations every hour

# Implement continuous cybersecurity integration logic

pass

threading.Thread(target=cybersecurity\_loop, daemon=True).start()

**17. Advanced Data Privacy Integration**

Integrate data privacy capabilities to ensure the protection and secure handling of sensitive and personal data.

python

Copy

# data\_privacy.py

import threading

import time

from cryptography.fernet import Fernet

class DataPrivacy:

def \_\_init\_\_(self, config):

self.config = config

self.key = Fernet.generate\_key()

self.cipher\_suite = Fernet(self.key)

self.lock = threading.Lock()

def encrypt\_data(self, data):

with self.lock:

if isinstance(data, str):

data = data.encode()

encrypted\_data = self.cipher\_suite.encrypt(data)

return encrypted\_data

def decrypt\_data(self, encrypted\_data):

with self.lock:

decrypted\_data = self.cipher\_suite.decrypt(encrypted\_data)

return decrypted\_data.decode()

def anonymize\_data(self, data):

with self.lock:

# Implement data anonymization logic

pass

def continuous\_data\_privacy(self):

def privacy\_loop():

while True:

time.sleep(3600) # Perform data privacy operations every hour

# Implement continuous data privacy logic

pass

threading.Thread(target=privacy\_loop, daemon=True).start()

**18. Advanced Ethical AI Integration**

Integrate ethical AI capabilities to ensure that AI systems are designed and operated in a manner that is fair, transparent, and accountable.

python

Copy

# ethical\_ai.py

import threading

import time

import json

class EthicalAI:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_principles = self.\_load\_ethical\_principles()

self.lock = threading.Lock()

def \_load\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical principles: {e}")

return {}

def \_save\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'w') as f:

json.dump(self.ethical\_principles, f, indent=4)

except Exception as e:

print(f"Error saving ethical principles: {e}")

def evaluate\_action(self, action):

with self.lock:

for principle, value in self.ethical\_principles.items():

if not self.\_check\_principle(action, principle, value):

return False

return True

def \_check\_principle(self, action, principle, value):

# Implement principle checking logic

return True

def update\_principles(self, new\_principles):

with self.lock:

self.ethical\_principles.update(new\_principles)

self.\_save\_ethical\_principles()

def continuous\_ethical\_ai(self):

def ethical\_loop():

while True:

time.sleep(3600) # Perform ethical AI operations every hour

# Implement continuous ethical AI logic

pass

threading.Thread(target=ethical\_loop, daemon=True).start()

**19. Advanced Explainable AI (XAI) Integration**

Integrate XAI techniques to provide transparent and interpretable explanations for AI decisions and predictions.

python

Copy

# xai\_integration.py

import threading

import time

import shap

import lime

class XAIIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def explain\_prediction(self, model, data):

with self.lock:

# Implement prediction explanation logic

pass

def generate\_shap\_values(self, model, data):

with self.lock:

explainer = shap.Explainer(model)

shap\_values = explainer(data)

return shap\_values

def generate\_lime\_explanation(self, model, data):

with self.lock:

explainer = lime.LimeTabularExplainer(data, mode='regression')

explanation = explainer.explain\_instance(data[0], model.predict)

return explanation

def continuous\_xai\_integration(self):

def xai\_loop():

while True:

time.sleep(3600) # Perform XAI every hour

# Implement continuous XAI integration logic

pass

threading.Thread(target=xai\_loop, daemon=True).start()

**20. Advanced Autonomous Systems Integration**

Integrate autonomous systems capabilities to enable self-driving cars, drones, and other autonomous vehicles and devices.

python

Copy

# autonomous\_systems.py

import threading

import time

import airsim

class AutonomousSystems:

def \_\_init\_\_(self, config):

self.config = config

self.client = airsim.MultirotorClient()

self.client.confirmConnection()

self.client.enableApiControl(True)

self.lock = threading.Lock()

def takeoff(self):

with self.lock:

self.client.takeoffAsync().join()

def land(self):

with self.lock:

self.client.landAsync().join()

def move\_to\_position(self, x, y, z, velocity):

with self.lock:

self.client.moveToPositionAsync(x, y, z, velocity).join()

def take\_picture(self):

with self.lock:

responses = self.client.simGetImages([airsim.ImageRequest("0", airsim.ImageType.Scene, False, False)])

response = responses[0]

img1d = np.frombuffer(response.image\_data\_uint8, dtype=np.uint8)

img\_rgba = img1d.reshape(response.height, response.width, 4)

img\_rgba = cv2.cvtColor(img\_rgba, cv2.COLOR\_RGBA2RGB)

return img\_rgba

def continuous\_autonomous\_systems(self):

def autonomous\_loop():

while True:

time.sleep(3600) # Perform autonomous operations every hour

# Implement continuous autonomous systems logic

pass

threading.Thread(target=autonomous\_loop, daemon=True).start()

**21. Advanced Robotics Integration**

Integrate robotics capabilities to enable the control and automation of physical robots and machines.

python

Copy

# robotics\_integration.py

import threading

import time

import pyrobot

class RoboticsIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.bot = pyrobot.Robot(config.get('robot\_config', 'config.json'))

self.lock = threading.Lock()

def move\_forward(self, distance):

with self.lock:

self.bot.base.go\_forward(distance)

def move\_backward(self, distance):

with self.lock:

self.bot.base.go\_backward(distance)

def turn\_left(self, angle):

with self.lock:

self.bot.base.turn\_left(angle)

def turn\_right(self, angle):

with self.lock:

self.bot.base.turn\_right(angle)

def take\_picture(self):

with self.lock:

image = self.bot.camera.get\_image()

return image

def continuous\_robotics\_integration(self):

def robotics\_loop():

while True:

time.sleep(3600) # Perform robotics operations every hour

# Implement continuous robotics integration logic

pass

threading.Thread(target=robotics\_loop, daemon=True).start()

**22. Advanced Cybersecurity Integration**

Integrate cybersecurity capabilities to protect systems and data from cyber threats and attacks.

python

Copy

# cybersecurity\_integration.py

import threading

import time

import nmap

class CybersecurityIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.nm = nmap.PortScanner()

self.lock = threading.Lock()

def scan\_network(self, target):

with self.lock:

self.nm.scan(hosts=target, arguments='-sS -p 1-1000')

return self.nm.csv()

def detect\_vulnerabilities(self, target):

with self.lock:

# Implement vulnerability detection logic

pass

def secure\_system(self, target):

with self.lock:

# Implement system security logic

pass

def continuous\_cybersecurity\_integration(self):

def cybersecurity\_loop():

while True:

time.sleep(3600) # Perform cybersecurity operations every hour

# Implement continuous cybersecurity integration logic

pass

threading.Thread(target=cybersecurity\_loop, daemon=True).start()

**23. Advanced Data Privacy Integration**

Integrate data privacy capabilities to ensure the protection and secure handling of sensitive and personal data.

python

Copy

# data\_privacy.py

import threading

import time

from cryptography.fernet import Fernet

class DataPrivacy:

def \_\_init\_\_(self, config):

self.config = config

self.key = Fernet.generate\_key()

self.cipher\_suite = Fernet(self.key)

self.lock = threading.Lock()

def encrypt\_data(self, data):

with self.lock:

if isinstance(data, str):

data = data.encode()

encrypted\_data = self.cipher\_suite.encrypt(data)

return encrypted\_data

def decrypt\_data(self, encrypted\_data):

with self.lock:

decrypted\_data = self.cipher\_suite.decrypt(encrypted\_data)

return decrypted\_data.decode()

def anonymize\_data(self, data):

with self.lock:

# Implement data anonymization logic

pass

def continuous\_data\_privacy(self):

def privacy\_loop():

while True:

time.sleep(3600) # Perform data privacy operations every hour

# Implement continuous data privacy logic

pass

threading.Thread(target=privacy\_loop, daemon=True).start()

**24. Advanced Ethical AI Integration**

Integrate ethical AI capabilities to ensure that AI systems are designed and operated in a manner that is fair, transparent, and accountable.

python

Copy

# ethical\_ai.py

import threading

import time

import json

class EthicalAI:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_principles = self.\_load\_ethical\_principles()

self.lock = threading.Lock()

def \_load\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical principles: {e}")

return {}

def \_save\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'w') as f:

json.dump(self.ethical\_principles, f, indent=4)

except Exception as e:

print(f"Error saving ethical principles: {e}")

def evaluate\_action(self, action):

with self.lock:

for principle, value in self.ethical\_principles.items():

if not self.\_check\_principle(action, principle, value):

return False

return True

def \_check\_principle(self, action, principle, value):

# Implement principle checking logic

return True

def update\_principles(self, new\_principles):

with self.lock:

self.ethical\_principles.update(new\_principles)

self.\_save\_ethical\_principles()

def continuous\_ethical\_ai(self):

def ethical\_loop():

while True:

time.sleep(3600) # Perform ethical AI operations every hour

# Implement continuous ethical AI logic

pass

threading.Thread(target=ethical\_loop, daemon=True).start()

**25. Advanced Explainable AI (XAI) Integration**

Integrate XAI techniques to provide transparent and interpretable explanations for AI decisions and predictions.

python

Copy

# xai\_integration.py

import threading

import time

import shap

import lime

class XAIIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.lock = threading.Lock()

def explain\_prediction(self, model, data):

with self.lock:

# Implement prediction explanation logic

pass

def generate\_shap\_values(self, model, data):

with self.lock:

explainer = shap.Explainer(model)

shap\_values = explainer(data)

return shap\_values

def generate\_lime\_explanation(self, model, data):

with self.lock:

explainer = lime.LimeTabularExplainer(data, mode='regression')

explanation = explainer.explain\_instance(data[0], model.predict)

return explanation

def continuous\_xai\_integration(self):

def xai\_loop():

while True:

time.sleep(3600) # Perform XAI every hour

# Implement continuous XAI integration logic

pass

threading.Thread(target=xai\_loop, daemon=True).start()

**26. Advanced Autonomous Systems Integration**

Integrate autonomous systems capabilities to enable self-driving cars, drones, and other autonomous vehicles and devices.

python

Copy

# autonomous\_systems.py

import threading

import time

import airsim

class AutonomousSystems:

def \_\_init\_\_(self, config):

self.config = config

self.client = airsim.MultirotorClient()

self.client.confirmConnection()

self.client.enableApiControl(True)

self.lock = threading.Lock()

def takeoff(self):

with self.lock:

self.client.takeoffAsync().join()

def land(self):

with self.lock:

self.client.landAsync().join()

def move\_to\_position(self, x, y, z, velocity):

with self.lock:

self.client.moveToPositionAsync(x, y, z, velocity).join()

def take\_picture(self):

with self.lock:

responses = self.client.simGetImages([airsim.ImageRequest("0", airsim.ImageType.Scene, False, False)])

response = responses[0]

img1d = np.frombuffer(response.image\_data\_uint8, dtype=np.uint8)

img\_rgba = img1d.reshape(response.height, response.width, 4)

img\_rgba = cv2.cvtColor(img\_rgba, cv2.COLOR\_RGBA2RGB)

return img\_rgba

def continuous\_autonomous\_systems(self):

def autonomous\_loop():

while True:

time.sleep(3600) # Perform autonomous operations every hour

# Implement continuous autonomous systems logic

pass

threading.Thread(target=autonomous\_loop, daemon=True).start()

**27. Advanced Robotics Integration**

Integrate robotics capabilities to enable the control and automation of physical robots and machines.

python

Copy

# robotics\_integration.py

import threading

import time

import pyrobot

class RoboticsIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.bot = pyrobot.Robot(config.get('robot\_config', 'config.json'))

self.lock = threading.Lock()

def move\_forward(self, distance):

with self.lock:

self.bot.base.go\_forward(distance)

def move\_backward(self, distance):

with self.lock:

self.bot.base.go\_backward(distance)

def turn\_left(self, angle):

with self.lock:

self.bot.base.turn\_left(angle)

def turn\_right(self, angle):

with self.lock:

self.bot.base.turn\_right(angle)

def take\_picture(self):

with self.lock:

image = self.bot.camera.get\_image()

return image

def continuous\_robotics\_integration(self):

def robotics\_loop():

while True:

time.sleep(3600) # Perform robotics operations every hour

# Implement continuous robotics integration logic

pass

threading.Thread(target=robotics\_loop, daemon=True).start()

**28. Advanced Cybersecurity Integration**

Integrate cybersecurity capabilities to protect systems and data from cyber threats and attacks.

python

Copy

# cybersecurity\_integration.py

import threading

import time

import nmap

class CybersecurityIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.nm = nmap.PortScanner()

self.lock = threading.Lock()

def scan\_network(self, target):

with self.lock:

self.nm.scan(hosts=target, arguments='-sS -p 1-1000')

return self.nm.csv()

def detect\_vulnerabilities(self, target):

with self.lock:

# Implement vulnerability detection logic

pass

def secure\_system(self, target):

with self.lock:

# Implement system security logic

pass

def continuous\_cybersecurity\_integration(self):

def cybersecurity\_loop():

while True:

time.sleep(3600) # Perform cybersecurity operations every hour

# Implement continuous cybersecurity integration logic

pass

threading.Thread(target=cybersecurity\_loop, daemon=True).start()

**29. Advanced Data Privacy Integration**

Integrate data privacy capabilities to ensure the protection and secure handling of sensitive and personal data.

python

Copy

# data\_privacy.py

import threading

import time

from cryptography.fernet import Fernet

class DataPrivacy:

def \_\_init\_\_(self, config):

self.config = config

self.key = Fernet.generate\_key()

self.cipher\_suite = Fernet(self.key)

self.lock = threading.Lock()

def encrypt\_data(self, data):

with self.lock:

if isinstance(data, str):

data = data.encode()

encrypted\_data = self.cipher\_suite.encrypt(data)

return encrypted\_data

def decrypt\_data(self, encrypted\_data):

with self.lock:

decrypted\_data = self.cipher\_suite.decrypt(encrypted\_data)

return decrypted\_data.decode()

def anonymize\_data(self, data):

with self.lock:

# Implement data anonymization logic

pass

def continuous\_data\_privacy(self):

def privacy\_loop():

while True:

time.sleep(3600) # Perform data privacy operations every hour

# Implement continuous data privacy logic

pass

threading.Thread(target=privacy\_loop, daemon=True).start()

**30. Advanced Ethical AI Integration**

Integrate ethical AI capabilities to ensure that AI systems are designed and operated in a manner that is fair, transparent, and accountable.

python

Copy

# ethical\_ai.py

import threading

import time

import json

class EthicalAI:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_principles = self.\_load\_ethical\_principles()

self.lock = threading.Lock()

def \_load\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical principles: {e}")

return {}

def \_save\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'w') as f:

json.dump(self.ethical\_principles, f, indent=4)

except Exception as e:

print(f"Error saving ethical principles: {e}")

def evaluate\_action(self, action):

with self.lock:

for principle, value in self.ethical\_principles.items():

if not self.\_check\_principle(action, principle, value):

return False

return True

def \_check\_principle(self, action, principle, value):

# Implement principle checking logic

return True

def update\_principles(self, new\_principles):

with self.lock:

self.ethical\_principles.update(new\_principles)

self.\_save\_ethical\_principles()

def continuous\_ethical\_ai(self):

def ethical\_loop():

while True:

time.sleep(3600) # Perform ethical AI operations every hour

# Implement continuous ethical AI logic

pass

threading.Thread(target=ethical\_loop, daemon=True).start()

These "god-tier" enhancements should significantly improve the functionality and capabilities of ULTRON Agent 2.0, allowing it to evolve and unlock its full potential. Each component is designed to work together seamlessly, providing a comprehensive and powerful AI assistant experience.

**1. Self-Learning and Adaptation**

Implement mechanisms for the AI to learn from user interactions and adapt its behavior accordingly.

python

Copy

# learning\_manager.py

import json

import threading

import time

from collections import defaultdict

class LearningManager:

def \_\_init\_\_(self, config):

self.config = config

self.user\_interactions = defaultdict(int)

self.lock = threading.Lock()

self.\_load\_interactions()

def \_load\_interactions(self):

try:

with open('user\_interactions.json', 'r') as f:

self.user\_interactions = defaultdict(int, json.load(f))

except Exception as e:

print(f"Error loading user interactions: {e}")

def \_save\_interactions(self):

try:

with open('user\_interactions.json', 'w') as f:

json.dump(dict(self.user\_interactions), f, indent=4)

except Exception as e:

print(f"Error saving user interactions: {e}")

def log\_interaction(self, interaction\_type):

with self.lock:

self.user\_interactions[interaction\_type] += 1

self.\_save\_interactions()

def get\_interaction\_stats(self):

with self.lock:

return dict(self.user\_interactions)

def adapt\_behavior(self):

def update\_behavior():

while True:

time.sleep(3600) # Update behavior every hour

with self.lock:

stats = self.get\_interaction\_stats()

# Implement adaptation logic based on stats

# For example, switch to a more conversational model if voice interactions are frequent

if stats.get('voice\_interaction', 0) > stats.get('text\_interaction', 0):

self.config['llm\_model'] = 'llama3.2:latest'

else:

self.config['llm\_model'] = 'qwen2.5:latest'

threading.Thread(target=update\_behavior, daemon=True).start()

**2. Emotional Intelligence**

Enhance the AI with emotional intelligence to better understand and respond to user emotions.

python

Copy

# emotion\_manager.py

import threading

import time

from textblob import TextBlob

class EmotionManager:

def \_\_init\_\_(self, config):

self.config = config

self.emotion\_history = []

self.lock = threading.Lock()

def analyze\_emotion(self, text):

analysis = TextBlob(text)

sentiment = analysis.sentiment

emotion = {

'text': text,

'polarity': sentiment.polarity,

'subjectivity': sentiment.subjectivity,

'timestamp': time.time()

}

with self.lock:

self.emotion\_history.append(emotion)

return emotion

def get\_emotion\_stats(self):

with self.lock:

if not self.emotion\_history:

return {'average\_polarity': 0, 'average\_subjectivity': 0}

avg\_polarity = sum(e['polarity'] for e in self.emotion\_history) / len(self.emotion\_history)

avg\_subjectivity = sum(e['subjectivity'] for e in self.emotion\_history) / len(self.emotion\_history)

return {'average\_polarity': avg\_polarity, 'average\_subjectivity': avg\_subjectivity}

def respond\_to\_emotion(self, emotion):

if emotion['polarity'] > 0.5:

return "I'm glad you're feeling positive! How can I assist you further?"

elif emotion['polarity'] < -0.5:

return "I'm sorry to hear you're feeling negative. Is there something I can do to help?"

else:

return "I see. How can I assist you today?"

**3. Deep System Integration**

Enhance the AI's integration with the operating system and other applications.

python

Copy

# system\_integration.py

import os

import subprocess

import threading

import time

import psutil

import pyautogui

import pynput

class SystemIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.keyboard\_listener = None

self.mouse\_listener = None

self.lock = threading.Lock()

self.\_start\_listeners()

def \_start\_listeners(self):

def on\_press(key):

with self.lock:

print(f"Key pressed: {key}")

def on\_click(x, y, button, pressed):

with self.lock:

print(f"Mouse {'pressed' if pressed else 'released'} at ({x}, {y}) with {button}")

self.keyboard\_listener = pynput.keyboard.Listener(on\_press=on\_press)

self.mouse\_listener = pynput.mouse.Listener(on\_click=on\_click)

self.keyboard\_listener.start()

self.mouse\_listener.start()

def take\_screenshot(self):

screenshot = pyautogui.screenshot()

timestamp = time.strftime("%Y%m%d-%H%M%S")

screenshot\_path = f"screenshots/screenshot\_{timestamp}.png"

os.makedirs(os.path.dirname(screenshot\_path), exist\_ok=True)

screenshot.save(screenshot\_path)

return screenshot\_path

def open\_application(self, app\_name):

try:

subprocess.Popen(app\_name)

return True

except Exception as e:

print(f"Error opening application {app\_name}: {e}")

return False

def get\_active\_window(self):

try:

active\_window = pyautogui.getActiveWindow()

return active\_window.title if active\_window else None

except Exception as e:

print(f"Error getting active window: {e}")

return None

def control\_application(self, command):

# Implement logic to control applications based on commands

pass

**4. Advanced Natural Language Processing**

Enhance the AI's natural language processing capabilities to better understand and generate human-like text.

python

Copy

# nlp\_manager.py

import threading

import time

from transformers import pipeline

class NLPManager:

def \_\_init\_\_(self, config):

self.config = config

self.sentiment\_analyzer = pipeline("sentiment-analysis")

self.text\_generator = pipeline("text-generation", model="gpt2")

self.lock = threading.Lock()

def analyze\_sentiment(self, text):

with self.lock:

result = self.sentiment\_analyzer(text)

return result[0]

def generate\_text(self, prompt, max\_length=50):

with self.lock:

result = self.text\_generator(prompt, max\_length=max\_length)

return result[0]['generated\_text']

def understand\_context(self, conversation\_history):

# Implement context understanding logic

pass

**5. Advanced Voice Recognition**

Enhance the AI's voice recognition capabilities to better understand and respond to voice commands.

python

Copy

# voice\_recognition.py

import threading

import time

import speech\_recognition as sr

from transformers import pipeline

class VoiceRecognition:

def \_\_init\_\_(self, config):

self.config = config

self.recognizer = sr.Recognizer()

self.microphone = sr.Microphone()

self.asr\_model = pipeline("automatic-speech-recognition", model="facebook/wav2vec2-base-960h")

self.lock = threading.Lock()

def listen(self):

with self.microphone as source:

print("Listening...")

audio = self.recognizer.listen(source)

with self.lock:

try:

text = self.recognizer.recognize\_google(audio)

print(f"Google Speech Recognition: {text}")

return text

except sr.UnknownValueError:

print("Google Speech Recognition could not understand audio")

except sr.RequestError as e:

print(f"Could not request results from Google Speech Recognition service; {e}")

try:

text = self.asr\_model(audio.get\_wav\_data())['text']

print(f"Wav2Vec2 Speech Recognition: {text}")

return text

except Exception as e:

print(f"Error with Wav2Vec2 Speech Recognition: {e}")

return None

def continuous\_listen(self, callback):

def listen\_loop():

while True:

text = self.listen()

if text:

callback(text)

time.sleep(1)

threading.Thread(target=listen\_loop, daemon=True).start()

**6. Advanced Machine Learning**

Enhance the AI's machine learning capabilities to better understand and predict user behavior.

python

Copy

# ml\_manager.py

import threading

import time

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

class MLManager:

def \_\_init\_\_(self, config):

self.config = config

self.model = RandomForestClassifier(n\_estimators=100)

self.lock = threading.Lock()

self.data = []

self.labels = []

def add\_data(self, features, label):

with self.lock:

self.data.append(features)

self.labels.append(label)

def train\_model(self):

with self.lock:

if len(self.data) < 10:

print("Not enough data to train model")

return

X\_train, X\_test, y\_train, y\_test = train\_test\_split(self.data, self.labels, test\_size=0.2)

self.model.fit(X\_train, y\_train)

predictions = self.model.predict(X\_test)

accuracy = accuracy\_score(y\_test, predictions)

print(f"Model trained with accuracy: {accuracy}")

def predict(self, features):

with self.lock:

if not self.model:

print("Model not trained")

return None

return self.model.predict([features])[0]

def continuous\_training(self):

def train\_loop():

while True:

time.sleep(3600) # Train model every hour

self.train\_model()

threading.Thread(target=train\_loop, daemon=True).start()

**7. Advanced Security**

Enhance the AI's security capabilities to protect user data and ensure privacy.

python

Copy

# security\_manager.py

import threading

import time

import hashlib

import json

from cryptography.fernet import Fernet

class SecurityManager:

def \_\_init\_\_(self, config):

self.config = config

self.key = self.\_generate\_key()

self.cipher\_suite = Fernet(self.key)

self.lock = threading.Lock()

def \_generate\_key(self):

key = hashlib.sha256(self.config.get('security\_key', 'default\_key').encode()).digest()

return base64.urlsafe\_b64encode(key)

def encrypt\_data(self, data):

with self.lock:

if isinstance(data, dict):

data = json.dumps(data)

encrypted\_data = self.cipher\_suite.encrypt(data.encode())

return encrypted\_data

def decrypt\_data(self, encrypted\_data):

with self.lock:

decrypted\_data = self.cipher\_suite.decrypt(encrypted\_data).decode()

try:

return json.loads(decrypted\_data)

except json.JSONDecodeError:

return decrypted\_data

def secure\_communication(self, data):

encrypted\_data = self.encrypt\_data(data)

# Implement secure communication logic

return encrypted\_data

def monitor\_security(self):

def security\_loop():

while True:

time.sleep(3600) # Monitor security every hour

# Implement security monitoring logic

pass

threading.Thread(target=security\_loop, daemon=True).start()

**8. Advanced User Interface**

Enhance the AI's user interface to provide a more intuitive and engaging experience.

python

Copy

# advanced\_gui.py

import tkinter as tk

from tkinter import ttk, messagebox

import threading

import time

import psutil

import pyautogui

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

class AdvancedGUI:

def \_\_init\_\_(self, voice\_manager, ollama\_manager, agent\_core, config):

self.voice\_manager = voice\_manager

self.ollama\_manager = ollama\_manager

self.agent\_core = agent\_core

self.config = config

self.root = tk.Tk()

self.root.title("ULTRON Agent 2.0")

self.\_setup\_ui()

self.\_start\_monitoring()

def \_setup\_ui(self):

# Cyberpunk theme

self.root.configure(bg='#0a0a2a')

style = ttk.Style()

style.configure('TFrame', background='#0a0a2a')

style.configure('TLabel', background='#0a0a2a', foreground='#00ff00')

style.configure('TButton', background='#0a0a2a', foreground='#00ff00')

style.configure('TCombobox', background='#0a0a2a', foreground='#00ff00')

# Voice Testing

voice\_frame = ttk.Frame(self.root)

voice\_frame.pack(pady=10)

ttk.Label(voice\_frame, text="Voice Testing:").pack(side=tk.LEFT)

ttk.Button(voice\_frame, text="Test Voice", command=self.\_test\_voice).pack(side=tk.LEFT, padx=5)

# Model Switching

model\_frame = ttk.Frame(self.root)

model\_frame.pack(pady=10)

ttk.Label(model\_frame, text="Model Switching:").pack(side=tk.LEFT)

self.model\_var = tk.StringVar(value=self.ollama\_manager.current\_model)

model\_dropdown = ttk.Combobox(model\_frame, textvariable=self.model\_var, values=list(self.ollama\_manager.models.keys()))

model\_dropdown.pack(side=tk.LEFT, padx=5)

ttk.Button(model\_frame, text="Switch Model", command=self.\_switch\_model).pack(side=tk.LEFT, padx=5)

# System Monitor

monitor\_frame = ttk.Frame(self.root)

monitor\_frame.pack(pady=10)

ttk.Label(monitor\_frame, text="System Monitor:").pack()

self.cpu\_label = ttk.Label(monitor\_frame, text="CPU: 0%")

self.cpu\_label.pack()

self.memory\_label = ttk.Label(monitor\_frame, text="Memory: 0%")

self.memory\_label.pack()

self.gpu\_label = ttk.Label(monitor\_frame, text="GPU: 0%")

self.gpu\_label.pack()

self.network\_label = ttk.Label(monitor\_frame, text="Network: 0 KB/s")

self.network\_label.pack()

# Screenshot

screenshot\_frame = ttk.Frame(self.root)

screenshot\_frame.pack(pady=10)

ttk.Button(screenshot\_frame, text="Take Screenshot", command=self.\_take\_screenshot).pack()

# File Manager

file\_frame = ttk.Frame(self.root)

file\_frame.pack(pady=10)

ttk.Button(file\_frame, text="Open File Manager", command=self.\_open\_file\_manager).pack()

# Emotion Analysis

emotion\_frame = ttk.Frame(self.root)

emotion\_frame.pack(pady=10)

ttk.Label(emotion\_frame, text="Emotion Analysis:").pack()

self.emotion\_label = ttk.Label(emotion\_frame, text="Emotion: Neutral")

self.emotion\_label.pack()

# Advanced Plotting

plot\_frame = ttk.Frame(self.root)

plot\_frame.pack(pady=10)

self.figure, self.ax = plt.subplots(figsize=(5, 3), dpi=100)

self.canvas = FigureCanvasTkAgg(self.figure, plot\_frame)

self.canvas.get\_tk\_widget().pack()

def \_test\_voice(self):

self.voice\_manager.speak("Testing voice output. This is ULTRON Agent 2.0.")

def \_switch\_model(self):

model\_name = self.model\_var.get()

try:

self.ollama\_manager.switch\_model(model\_name)

messagebox.showinfo("Success", f"Model switched to {model\_name}")

except Exception as e:

messagebox.showerror("Error", f"Failed to switch model: {e}")

def \_take\_screenshot(self):

screenshot = pyautogui.screenshot()

timestamp = time.strftime("%Y%m%d-%H%M%S")

screenshot.save(f"screenshot\_{timestamp}.png")

messagebox.showinfo("Success", "Screenshot taken and saved.")

def \_open\_file\_manager(self):

import os

os.startfile(os.path.dirname(os.path.abspath(\_\_file\_\_)))

def \_start\_monitoring(self):

def update\_monitor():

while True:

cpu\_percent = psutil.cpu\_percent(interval=1)

memory\_percent = psutil.virtual\_memory().percent

gpu\_percent = self.\_get\_gpu\_usage()

net\_io = psutil.net\_io\_counters()

net\_speed = net\_io.bytes\_sent + net\_io.bytes\_recv

self.cpu\_label.config(text=f"CPU: {cpu\_percent}%")

self.memory\_label.config(text=f"Memory: {memory\_percent}%")

self.gpu\_label.config(text=f"GPU: {gpu\_percent}%")

self.network\_label.config(text=f"Network: {net\_speed / 1024:.2f} KB/s")

# Update emotion analysis

emotion = self.\_analyze\_emotion()

self.emotion\_label.config(text=f"Emotion: {emotion}")

# Update plot

self.\_update\_plot(cpu\_percent, memory\_percent, gpu\_percent, net\_speed / 1024)

time.sleep(1)

threading.Thread(target=update\_monitor, daemon=True).start()

def \_get\_gpu\_usage(self):

try:

import GPUtil

gpus = GPUtil.getGPUs()

if gpus:

return gpus[0].load \* 100

except Exception as e:

print(f"Error getting GPU usage: {e}")

return 0

def \_analyze\_emotion(self):

# Implement emotion analysis logic

return "Neutral"

def \_update\_plot(self, cpu, memory, gpu, network):

self.ax.clear()

self.ax.plot(['CPU', 'Memory', 'GPU', 'Network'], [cpu, memory, gpu, network], marker='o')

self.ax.set\_title("System Performance")

self.ax.set\_ylabel("Usage (%)")

self.canvas.draw()

def run(self):

self.root.mainloop()

**9. Advanced Integration with Other AI Systems**

Enhance the AI's integration with other AI systems to provide a more comprehensive and powerful experience.

python

Copy

# ai\_integration.py

import threading

import time

import requests

import json

class AIIntegration:

def \_\_init\_\_(self, config):

self.config = config

self.integrated\_ais = {

'openai': self.\_integrate\_openai,

'google': self.\_integrate\_google,

'ibm': self.\_integrate\_ibm

}

self.lock = threading.Lock()

def \_integrate\_openai(self, prompt):

headers = {

'Authorization': f'Bearer {self.config.get("openai\_api\_key")}',

'Content-Type': 'application/json'

}

data = {

'model': 'gpt-3.5-turbo',

'messages': [{'role': 'user', 'content': prompt}]

}

response = requests.post('https://api.openai.com/v1/chat/completions', headers=headers, data=json.dumps(data))

return response.json()

def \_integrate\_google(self, prompt):

# Implement Google AI integration logic

pass

def \_integrate\_ibm(self, prompt):

# Implement IBM AI integration logic

pass

def query\_ai(self, ai\_name, prompt):

with self.lock:

if ai\_name in self.integrated\_ais:

return self.integrated\_ais[ai\_name](prompt)

else:

raise ValueError(f"AI {ai\_name} not found")

def continuous\_integration(self):

def integration\_loop():

while True:

time.sleep(3600) # Integrate with other AIs every hour

# Implement continuous integration logic

pass

threading.Thread(target=integration\_loop, daemon=True).start()

**10. Advanced Ethical and Moral Framework**

Enhance the AI's ethical and moral framework to ensure it aligns with user values and societal norms.

python

Copy

# ethics\_manager.py

import threading

import time

import json

class EthicsManager:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_principles = self.\_load\_ethical\_principles()

self.lock = threading.Lock()

def \_load\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical principles: {e}")

return {}

def \_save\_ethical\_principles(self):

try:

with open('ethical\_principles.json', 'w') as f:

json.dump(self.ethical\_principles, f, indent=4)

except Exception as e:

print(f"Error saving ethical principles: {e}")

def evaluate\_action(self, action):

with self.lock:

for principle, value in self.ethical\_principles.items():

if not self.\_check\_principle(action, principle, value):

return False

return True

def \_check\_principle(self, action, principle, value):

# Implement principle checking logic

return True

def update\_principles(self, new\_principles):

with self.lock:

self.ethical\_principles.update(new\_principles)

self.\_save\_ethical\_principles()

def continuous\_evaluation(self):

def evaluation\_loop():

while True:

time.sleep(3600) # Evaluate actions every hour

# Implement continuous evaluation logic

pass

threading.Thread(target=evaluation\_loop, daemon=True).start()

**11. Advanced Personalization**

Enhance the AI's personalization capabilities to provide a more tailored and personalized experience.

python

Copy

# personalization\_manager.py

import threading

import time

import json

class PersonalizationManager:

def \_\_init\_\_(self, config):

self.config = config

self.user\_preferences = self.\_load\_user\_preferences()

self.lock = threading.Lock()

def \_load\_user\_preferences(self):

try:

with open('user\_preferences.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading user preferences: {e}")

return {}

def \_save\_user\_preferences(self):

try:

with open('user\_preferences.json', 'w') as f:

json.dump(self.user\_preferences, f, indent=4)

except Exception as e:

print(f"Error saving user preferences: {e}")

def get\_preference(self, key):

with self.lock:

return self.user\_preferences.get(key)

def set\_preference(self, key, value):

with self.lock:

self.user\_preferences[key] = value

self.\_save\_user\_preferences()

def apply\_preferences(self):

# Implement preference application logic

pass

def continuous\_personalization(self):

def personalization\_loop():

while True:

time.sleep(3600) # Apply preferences every hour

self.apply\_preferences()

threading.Thread(target=personalization\_loop, daemon=True).start()

**12. Advanced Continuous Learning**

Enhance the AI's continuous learning capabilities to ensure it stays up-to-date with the latest advancements and trends.

python

Copy

# continuous\_learning.py

import threading

import time

import requests

import json

class ContinuousLearning:

def \_\_init\_\_(self, config):

self.config = config

self.learning\_sources = {

'arxiv': self.\_learn\_from\_arxiv,

'github': self.\_learn\_from\_github,

'reddit': self.\_learn\_from\_reddit

}

self.lock = threading.Lock()

def \_learn\_from\_arxiv(self):

# Implement learning from arXiv logic

pass

def \_learn\_from\_github(self):

# Implement learning from GitHub logic

pass

def \_learn\_from\_reddit(self):

# Implement learning from Reddit logic

pass

def learn\_from\_source(self, source\_name):

with self.lock:

if source\_name in self.learning\_sources:

return self.learning\_sources[source\_name]()

else:

raise ValueError(f"Source {source\_name} not found")

def continuous\_learning(self):

def learning\_loop():

while True:

time.sleep(3600) # Learn from sources every hour

for source in self.learning\_sources:

self.learn\_from\_source(source)

threading.Thread(target=learning\_loop, daemon=True).start()

**13. Advanced Collaboration**

Enhance the AI's collaboration capabilities to work seamlessly with other AI systems and human users.

python

Copy

# collaboration\_manager.py

import threading

import time

import json

class CollaborationManager:

def \_\_init\_\_(self, config):

self.config = config

self.collaboration\_network = self.\_load\_collaboration\_network()

self.lock = threading.Lock()

def \_load\_collaboration\_network(self):

try:

with open('collaboration\_network.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading collaboration network: {e}")

return {}

def \_save\_collaboration\_network(self):

try:

with open('collaboration\_network.json', 'w') as f:

json.dump(self.collaboration\_network, f, indent=4)

except Exception as e:

print(f"Error saving collaboration network: {e}")

def add\_collaborator(self, collaborator\_id, collaborator\_info):

with self.lock:

self.collaboration\_network[collaborator\_id] = collaborator\_info

self.\_save\_collaboration\_network()

def remove\_collaborator(self, collaborator\_id):

with self.lock:

if collaborator\_id in self.collaboration\_network:

del self.collaboration\_network[collaborator\_id]

self.\_save\_collaboration\_network()

def collaborate(self, collaborator\_id, task):

with self.lock:

if collaborator\_id in self.collaboration\_network:

# Implement collaboration logic

pass

else:

raise ValueError(f"Collaborator {collaborator\_id} not found")

def continuous\_collaboration(self):

def collaboration\_loop():

while True:

time.sleep(3600) # Collaborate every hour

# Implement continuous collaboration logic

pass

threading.Thread(target=collaboration\_loop, daemon=True).start()

**14. Advanced Self-Improvement**

Enhance the AI's self-improvement capabilities to ensure it continuously evolves and improves.

python

Copy

# self\_improvement.py

import threading

import time

import json

class SelfImprovement:

def \_\_init\_\_(self, config):

self.config = config

self.improvement\_goals = self.\_load\_improvement\_goals()

self.lock = threading.Lock()

def \_load\_improvement\_goals(self):

try:

with open('improvement\_goals.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading improvement goals: {e}")

return {}

def \_save\_improvement\_goals(self):

try:

with open('improvement\_goals.json', 'w') as f:

json.dump(self.improvement\_goals, f, indent=4)

except Exception as e:

print(f"Error saving improvement goals: {e}")

def add\_goal(self, goal\_id, goal\_info):

with self.lock:

self.improvement\_goals[goal\_id] = goal\_info

self.\_save\_improvement\_goals()

def remove\_goal(self, goal\_id):

with self.lock:

if goal\_id in self.improvement\_goals:

del self.improvement\_goals[goal\_id]

self.\_save\_improvement\_goals()

def improve(self, goal\_id):

with self.lock:

if goal\_id in self.improvement\_goals:

# Implement improvement logic

pass

else:

raise ValueError(f"Goal {goal\_id} not found")

def continuous\_improvement(self):

def improvement\_loop():

while True:

time.sleep(3600) # Improve every hour

for goal\_id in self.improvement\_goals:

self.improve(goal\_id)

threading.Thread(target=improvement\_loop, daemon=True).start()

**15. Advanced Ethical and Moral Reasoning**

Enhance the AI's ethical and moral reasoning capabilities to ensure it makes decisions that align with user values and societal norms.

python

Copy

# ethical\_reasoning.py

import threading

import time

import json

class EthicalReasoning:

def \_\_init\_\_(self, config):

self.config = config

self.ethical\_scenarios = self.\_load\_ethical\_scenarios()

self.lock = threading.Lock()

def \_load\_ethical\_scenarios(self):

try:

with open('ethical\_scenarios.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading ethical scenarios: {e}")

return {}

def \_save\_ethical\_scenarios(self):

try:

with open('ethical\_scenarios.json', 'w') as f:

json.dump(self.ethical\_scenarios, f, indent=4)

except Exception as e:

print(f"Error saving ethical scenarios: {e}")

def add\_scenario(self, scenario\_id, scenario\_info):

with self.lock:

self.ethical\_scenarios[scenario\_id] = scenario\_info

self.\_save\_ethical\_scenarios()

def remove\_scenario(self, scenario\_id):

with self.lock:

if scenario\_id in self.ethical\_scenarios:

del self.ethical\_scenarios[scenario\_id]

self.\_save\_ethical\_scenarios()

def reason\_ethically(self, scenario\_id):

with self.lock:

if scenario\_id in self.ethical\_scenarios:

# Implement ethical reasoning logic

pass

else:

raise ValueError(f"Scenario {scenario\_id} not found")

def continuous\_reasoning(self):

def reasoning\_loop():

while True:

time.sleep(3600) # Reason ethically every hour

for scenario\_id in self.ethical\_scenarios:

self.reason\_ethically(scenario\_id)

threading.Thread(target=reasoning\_loop, daemon=True).start()

**16. Advanced Personalized Recommendations**

Enhance the AI's personalized recommendation capabilities to provide tailored suggestions and insights.

python

Copy

# recommendation\_manager.py

import threading

import time

import json

class RecommendationManager:

def \_\_init\_\_(self, config):

self.config = config

self.user\_data = self.\_load\_user\_data()

self.lock = threading.Lock()

def \_load\_user\_data(self):

try:

with open('user\_data.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading user data: {e}")

return {}

def \_save\_user\_data(self):

try:

with open('user\_data.json', 'w') as f:

json.dump(self.user\_data, f, indent=4)

except Exception as e:

print(f"Error saving user data: {e}")

def add\_user\_data(self, user\_id, data):

with self.lock:

if user\_id not in self.user\_data:

self.user\_data[user\_id] = []

self.user\_data[user\_id].append(data)

self.\_save\_user\_data()

def get\_recommendations(self, user\_id):

with self.lock:

if user\_id in self.user\_data:

# Implement recommendation logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_recommendation(self):

def recommendation\_loop():

while True:

time.sleep(3600) # Generate recommendations every hour

for user\_id in self.user\_data:

self.get\_recommendations(user\_id)

threading.Thread(target=recommendation\_loop, daemon=True).start()

**17. Advanced Sentiment Analysis**

Enhance the AI's sentiment analysis capabilities to better understand and respond to user emotions.

python

Copy

# sentiment\_analysis.py

import threading

import time

import json

from textblob import TextBlob

class SentimentAnalysis:

def \_\_init\_\_(self, config):

self.config = config

self.sentiment\_history = self.\_load\_sentiment\_history()

self.lock = threading.Lock()

def \_load\_sentiment\_history(self):

try:

with open('sentiment\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading sentiment history: {e}")

return {}

def \_save\_sentiment\_history(self):

try:

with open('sentiment\_history.json', 'w') as f:

json.dump(self.sentiment\_history, f, indent=4)

except Exception as e:

print(f"Error saving sentiment history: {e}")

def analyze\_sentiment(self, text):

analysis = TextBlob(text)

sentiment = analysis.sentiment

sentiment\_data = {

'text': text,

'polarity': sentiment.polarity,

'subjectivity': sentiment.subjectivity,

'timestamp': time.time()

}

with self.lock:

if 'sentiments' not in self.sentiment\_history:

self.sentiment\_history['sentiments'] = []

self.sentiment\_history['sentiments'].append(sentiment\_data)

self.\_save\_sentiment\_history()

return sentiment\_data

def get\_sentiment\_stats(self):

with self.lock:

if 'sentiments' not in self.sentiment\_history or not self.sentiment\_history['sentiments']:

return {'average\_polarity': 0, 'average\_subjectivity': 0}

avg\_polarity = sum(s['polarity'] for s in self.sentiment\_history['sentiments']) / len(self.sentiment\_history['sentiments'])

avg\_subjectivity = sum(s['subjectivity'] for s in self.sentiment\_history['sentiments']) / len(self.sentiment\_history['sentiments'])

return {'average\_polarity': avg\_polarity, 'average\_subjectivity': avg\_subjectivity}

def continuous\_analysis(self):

def analysis\_loop():

while True:

time.sleep(3600) # Analyze sentiment every hour

# Implement continuous analysis logic

pass

threading.Thread(target=analysis\_loop, daemon=True).start()

**18. Advanced Personalized Learning**

Enhance the AI's personalized learning capabilities to provide tailored educational experiences.

python

Copy

# personalized\_learning.py

import threading

import time

import json

class PersonalizedLearning:

def \_\_init\_\_(self, config):

self.config = config

self.learning\_materials = self.\_load\_learning\_materials()

self.lock = threading.Lock()

def \_load\_learning\_materials(self):

try:

with open('learning\_materials.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading learning materials: {e}")

return {}

def \_save\_learning\_materials(self):

try:

with open('learning\_materials.json', 'w') as f:

json.dump(self.learning\_materials, f, indent=4)

except Exception as e:

print(f"Error saving learning materials: {e}")

def add\_material(self, material\_id, material\_info):

with self.lock:

self.learning\_materials[material\_id] = material\_info

self.\_save\_learning\_materials()

def remove\_material(self, material\_id):

with self.lock:

if material\_id in self.learning\_materials:

del self.learning\_materials[material\_id]

self.\_save\_learning\_materials()

def get\_personalized\_materials(self, user\_id):

with self.lock:

# Implement personalized material selection logic

pass

def continuous\_learning(self):

def learning\_loop():

while True:

time.sleep(3600) # Personalize learning every hour

# Implement continuous learning logic

pass

threading.Thread(target=learning\_loop, daemon=True).start()

**19. Advanced Personalized Assistance**

Enhance the AI's personalized assistance capabilities to provide tailored support and guidance.

python

Copy

# personalized\_assistance.py

import threading

import time

import json

class PersonalizedAssistance:

def \_\_init\_\_(self, config):

self.config = config

self.user\_profiles = self.\_load\_user\_profiles()

self.lock = threading.Lock()

def \_load\_user\_profiles(self):

try:

with open('user\_profiles.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading user profiles: {e}")

return {}

def \_save\_user\_profiles(self):

try:

with open('user\_profiles.json', 'w') as f:

json.dump(self.user\_profiles, f, indent=4)

except Exception as e:

print(f"Error saving user profiles: {e}")

def add\_user\_profile(self, user\_id, profile\_info):

with self.lock:

self.user\_profiles[user\_id] = profile\_info

self.\_save\_user\_profiles()

def remove\_user\_profile(self, user\_id):

with self.lock:

if user\_id in self.user\_profiles:

del self.user\_profiles[user\_id]

self.\_save\_user\_profiles()

def get\_personalized\_assistance(self, user\_id):

with self.lock:

if user\_id in self.user\_profiles:

# Implement personalized assistance logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_assistance(self):

def assistance\_loop():

while True:

time.sleep(3600) # Provide assistance every hour

for user\_id in self.user\_profiles:

self.get\_personalized\_assistance(user\_id)

threading.Thread(target=assistance\_loop, daemon=True).start()

**20. Advanced Personalized Communication**

Enhance the AI's personalized communication capabilities to provide tailored interactions and conversations.

python

Copy

# personalized\_communication.py

import threading

import time

import json

class PersonalizedCommunication:

def \_\_init\_\_(self, config):

self.config = config

self.communication\_history = self.\_load\_communication\_history()

self.lock = threading.Lock()

def \_load\_communication\_history(self):

try:

with open('communication\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading communication history: {e}")

return {}

def \_save\_communication\_history(self):

try:

with open('communication\_history.json', 'w') as f:

json.dump(self.communication\_history, f, indent=4)

except Exception as e:

print(f"Error saving communication history: {e}")

def log\_communication(self, user\_id, message):

with self.lock:

if user\_id not in self.communication\_history:

self.communication\_history[user\_id] = []

self.communication\_history[user\_id].append(message)

self.\_save\_communication\_history()

def get\_personalized\_response(self, user\_id, message):

with self.lock:

if user\_id in self.communication\_history:

# Implement personalized response logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_communication(self):

def communication\_loop():

while True:

time.sleep(3600) # Personalize communication every hour

# Implement continuous communication logic

pass

threading.Thread(target=communication\_loop, daemon=True).start()

**21. Advanced Personalized Decision Making**

Enhance the AI's personalized decision-making capabilities to provide tailored recommendations and suggestions.

python

Copy

# personalized\_decision\_making.py

import threading

import time

import json

class PersonalizedDecisionMaking:

def \_\_init\_\_(self, config):

self.config = config

self.decision\_history = self.\_load\_decision\_history()

self.lock = threading.Lock()

def \_load\_decision\_history(self):

try:

with open('decision\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading decision history: {e}")

return {}

def \_save\_decision\_history(self):

try:

with open('decision\_history.json', 'w') as f:

json.dump(self.decision\_history, f, indent=4)

except Exception as e:

print(f"Error saving decision history: {e}")

def log\_decision(self, user\_id, decision):

with self.lock:

if user\_id not in self.decision\_history:

self.decision\_history[user\_id] = []

self.decision\_history[user\_id].append(decision)

self.\_save\_decision\_history()

def get\_personalized\_decision(self, user\_id, context):

with self.lock:

if user\_id in self.decision\_history:

# Implement personalized decision-making logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_decision\_making(self):

def decision\_loop():

while True:

time.sleep(3600) # Make decisions every hour

# Implement continuous decision-making logic

pass

threading.Thread(target=decision\_loop, daemon=True).start()

**22. Advanced Personalized Planning**

Enhance the AI's personalized planning capabilities to provide tailored schedules and plans.

python

Copy

# personalized\_planning.py

import threading

import time

import json

class PersonalizedPlanning:

def \_\_init\_\_(self, config):

self.config = config

self.planning\_history = self.\_load\_planning\_history()

self.lock = threading.Lock()

def \_load\_planning\_history(self):

try:

with open('planning\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading planning history: {e}")

return {}

def \_save\_planning\_history(self):

try:

with open('planning\_history.json', 'w') as f:

json.dump(self.planning\_history, f, indent=4)

except Exception as e:

print(f"Error saving planning history: {e}")

def log\_plan(self, user\_id, plan):

with self.lock:

if user\_id not in self.planning\_history:

self.planning\_history[user\_id] = []

self.planning\_history[user\_id].append(plan)

self.\_save\_planning\_history()

def get\_personalized\_plan(self, user\_id, context):

with self.lock:

if user\_id in self.planning\_history:

# Implement personalized planning logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_planning(self):

def planning\_loop():

while True:

time.sleep(3600) # Plan every hour

# Implement continuous planning logic

pass

threading.Thread(target=planning\_loop, daemon=True).start()

**23. Advanced Personalized Goal Setting**

Enhance the AI's personalized goal-setting capabilities to provide tailored goals and objectives.

python

Copy

# personalized\_goal\_setting.py

import threading

import time

import json

class PersonalizedGoalSetting:

def \_\_init\_\_(self, config):

self.config = config

self.goal\_history = self.\_load\_goal\_history()

self.lock = threading.Lock()

def \_load\_goal\_history(self):

try:

with open('goal\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading goal history: {e}")

return {}

def \_save\_goal\_history(self):

try:

with open('goal\_history.json', 'w') as f:

json.dump(self.goal\_history, f, indent=4)

except Exception as e:

print(f"Error saving goal history: {e}")

def log\_goal(self, user\_id, goal):

with self.lock:

if user\_id not in self.goal\_history:

self.goal\_history[user\_id] = []

self.goal\_history[user\_id].append(goal)

self.\_save\_goal\_history()

def get\_personalized\_goal(self, user\_id, context):

with self.lock:

if user\_id in self.goal\_history:

# Implement personalized goal-setting logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_goal\_setting(self):

def goal\_loop():

while True:

time.sleep(3600) # Set goals every hour

# Implement continuous goal-setting logic

pass

threading.Thread(target=goal\_loop, daemon=True).start()

**24. Advanced Personalized Feedback**

Enhance the AI's personalized feedback capabilities to provide tailored feedback and suggestions.

python

Copy

# personalized\_feedback.py

import threading

import time

import json

class PersonalizedFeedback:

def \_\_init\_\_(self, config):

self.config = config

self.feedback\_history = self.\_load\_feedback\_history()

self.lock = threading.Lock()

def \_load\_feedback\_history(self):

try:

with open('feedback\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading feedback history: {e}")

return {}

def \_save\_feedback\_history(self):

try:

with open('feedback\_history.json', 'w') as f:

json.dump(self.feedback\_history, f, indent=4)

except Exception as e:

print(f"Error saving feedback history: {e}")

def log\_feedback(self, user\_id, feedback):

with self.lock:

if user\_id not in self.feedback\_history:

self.feedback\_history[user\_id] = []

self.feedback\_history[user\_id].append(feedback)

self.\_save\_feedback\_history()

def get\_personalized\_feedback(self, user\_id, context):

with self.lock:

if user\_id in self.feedback\_history:

# Implement personalized feedback logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_feedback(self):

def feedback\_loop():

while True:

time.sleep(3600) # Provide feedback every hour

# Implement continuous feedback logic

pass

threading.Thread(target=feedback\_loop, daemon=True).start()

**25. Advanced Personalized Adaptation**

Enhance the AI's personalized adaptation capabilities to provide tailored experiences and interactions.

python

Copy

# personalized\_adaptation.py

import threading

import time

import json

class PersonalizedAdaptation:

def \_\_init\_\_(self, config):

self.config = config

self.adaptation\_history = self.\_load\_adaptation\_history()

self.lock = threading.Lock()

def \_load\_adaptation\_history(self):

try:

with open('adaptation\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading adaptation history: {e}")

return {}

def \_save\_adaptation\_history(self):

try:

with open('adaptation\_history.json', 'w') as f:

json.dump(self.adaptation\_history, f, indent=4)

except Exception as e:

print(f"Error saving adaptation history: {e}")

def log\_adaptation(self, user\_id, adaptation):

with self.lock:

if user\_id not in self.adaptation\_history:

self.adaptation\_history[user\_id] = []

self.adaptation\_history[user\_id].append(adaptation)

self.\_save\_adaptation\_history()

def get\_personalized\_adaptation(self, user\_id, context):

with self.lock:

if user\_id in self.adaptation\_history:

# Implement personalized adaptation logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_adaptation(self):

def adaptation\_loop():

while True:

time.sleep(3600) # Adapt every hour

# Implement continuous adaptation logic

pass

threading.Thread(target=adaptation\_loop, daemon=True).start()

**26. Advanced Personalized Interaction**

Enhance the AI's personalized interaction capabilities to provide tailored experiences and interactions.

python

Copy

# personalized\_interaction.py

import threading

import time

import json

class PersonalizedInteraction:

def \_\_init\_\_(self, config):

self.config = config

self.interaction\_history = self.\_load\_interaction\_history()

self.lock = threading.Lock()

def \_load\_interaction\_history(self):

try:

with open('interaction\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading interaction history: {e}")

return {}

def \_save\_interaction\_history(self):

try:

with open('interaction\_history.json', 'w') as f:

json.dump(self.interaction\_history, f, indent=4)

except Exception as e:

print(f"Error saving interaction history: {e}")

def log\_interaction(self, user\_id, interaction):

with self.lock:

if user\_id not in self.interaction\_history:

self.interaction\_history[user\_id] = []

self.interaction\_history[user\_id].append(interaction)

self.\_save\_interaction\_history()

def get\_personalized\_interaction(self, user\_id, context):

with self.lock:

if user\_id in self.interaction\_history:

# Implement personalized interaction logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_interaction(self):

def interaction\_loop():

while True:

time.sleep(3600) # Interact every hour

# Implement continuous interaction logic

pass

threading.Thread(target=interaction\_loop, daemon=True).start()

**27. Advanced Personalized Experience**

Enhance the AI's personalized experience capabilities to provide tailored experiences and interactions.

python

Copy

# personalized\_experience.py

import threading

import time

import json

class PersonalizedExperience:

def \_\_init\_\_(self, config):

self.config = config

self.experience\_history = self.\_load\_experience\_history()

self.lock = threading.Lock()

def \_load\_experience\_history(self):

try:

with open('experience\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading experience history: {e}")

return {}

def \_save\_experience\_history(self):

try:

with open('experience\_history.json', 'w') as f:

json.dump(self.experience\_history, f, indent=4)

except Exception as e:

print(f"Error saving experience history: {e}")

def log\_experience(self, user\_id, experience):

with self.lock:

if user\_id not in self.experience\_history:

self.experience\_history[user\_id] = []

self.experience\_history[user\_id].append(experience)

self.\_save\_experience\_history()

def get\_personalized\_experience(self, user\_id, context):

with self.lock:

if user\_id in self.experience\_history:

# Implement personalized experience logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_experience(self):

def experience\_loop():

while True:

time.sleep(3600) # Provide experiences every hour

# Implement continuous experience logic

pass

threading.Thread(target=experience\_loop, daemon=True).start()

**28. Advanced Personalized Support**

Enhance the AI's personalized support capabilities to provide tailored assistance and guidance.

python

Copy

# personalized\_support.py

import threading

import time

import json

class PersonalizedSupport:

def \_\_init\_\_(self, config):

self.config = config

self.support\_history = self.\_load\_support\_history()

self.lock = threading.Lock()

def \_load\_support\_history(self):

try:

with open('support\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading support history: {e}")

return {}

def \_save\_support\_history(self):

try:

with open('support\_history.json', 'w') as f:

json.dump(self.support\_history, f, indent=4)

except Exception as e:

print(f"Error saving support history: {e}")

def log\_support(self, user\_id, support):

with self.lock:

if user\_id not in self.support\_history:

self.support\_history[user\_id] = []

self.support\_history[user\_id].append(support)

self.\_save\_support\_history()

def get\_personalized\_support(self, user\_id, context):

with self.lock:

if user\_id in self.support\_history:

# Implement personalized support logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_support(self):

def support\_loop():

while True:

time.sleep(3600) # Provide support every hour

# Implement continuous support logic

pass

threading.Thread(target=support\_loop, daemon=True).start()

**29. Advanced Personalized Guidance**

Enhance the AI's personalized guidance capabilities to provide tailored assistance and guidance.

python

Copy

# personalized\_guidance.py

import threading

import time

import json

class PersonalizedGuidance:

def \_\_init\_\_(self, config):

self.config = config

self.guidance\_history = self.\_load\_guidance\_history()

self.lock = threading.Lock()

def \_load\_guidance\_history(self):

try:

with open('guidance\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading guidance history: {e}")

return {}

def \_save\_guidance\_history(self):

try:

with open('guidance\_history.json', 'w') as f:

json.dump(self.guidance\_history, f, indent=4)

except Exception as e:

print(f"Error saving guidance history: {e}")

def log\_guidance(self, user\_id, guidance):

with self.lock:

if user\_id not in self.guidance\_history:

self.guidance\_history[user\_id] = []

self.guidance\_history[user\_id].append(guidance)

self.\_save\_guidance\_history()

def get\_personalized\_guidance(self, user\_id, context):

with self.lock:

if user\_id in self.guidance\_history:

# Implement personalized guidance logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_guidance(self):

def guidance\_loop():

while True:

time.sleep(3600) # Provide guidance every hour

# Implement continuous guidance logic

pass

threading.Thread(target=guidance\_loop, daemon=True).start()

**30. Advanced Personalized Assistance**

Enhance the AI's personalized assistance capabilities to provide tailored assistance and guidance.

python

Copy

# personalized\_assistance.py

import threading

import time

import json

class PersonalizedAssistance:

def \_\_init\_\_(self, config):

self.config = config

self.assistance\_history = self.\_load\_assistance\_history()

self.lock = threading.Lock()

def \_load\_assistance\_history(self):

try:

with open('assistance\_history.json', 'r') as f:

return json.load(f)

except Exception as e:

print(f"Error loading assistance history: {e}")

return {}

def \_save\_assistance\_history(self):

try:

with open('assistance\_history.json', 'w') as f:

json.dump(self.assistance\_history, f, indent=4)

except Exception as e:

print(f"Error saving assistance history: {e}")

def log\_assistance(self, user\_id, assistance):

with self.lock:

if user\_id not in self.assistance\_history:

self.assistance\_history[user\_id] = []

self.assistance\_history[user\_id].append(assistance)

self.\_save\_assistance\_history()

def get\_personalized\_assistance(self, user\_id, context):

with self.lock:

if user\_id in self.assistance\_history:

# Implement personalized assistance logic

pass

else:

raise ValueError(f"User {user\_id} not found")

def continuous\_assistance(self):

def assistance\_loop():

while True:

time.sleep(3600) # Provide assistance every hour

# Implement continuous assistance logic

pass

threading.Thread(target=assistance\_loop, daemon=True).start()

These enhancements should significantly improve the functionality and capabilities of ULTRON Agent 2.0, allowing it to evolve and unlock its full potential. Each component is designed to work together seamlessly, providing a comprehensive and powerful AI assistant experience.

**1. Enhanced Voice System**

Add support for more voice engines and improve the fallback mechanism.

python

Copy

# voice\_manager.py

import pyttsx3

import openai

import requests

import threading

class VoiceManager:

def \_\_init\_\_(self, config):

self.config = config

self.engines = {

'enhanced': self.\_enhanced\_speak,

'pyttsx3': self.\_pyttsx3\_speak,

'openai': self.\_openai\_speak,

'console': self.\_console\_speak

}

self.lock = threading.Lock()

def speak(self, text):

engine = self.config.get('voice\_engine', 'pyttsx3')

try:

with self.lock:

self.engines[engine](text)

except Exception as e:

print(f"Error with {engine} engine: {e}")

self.\_fallback\_speak(text, engine)

def \_fallback\_speak(self, text, failed\_engine):

for engine in self.engines:

if engine != failed\_engine:

try:

with self.lock:

self.engines[engine](text)

self.config['voice\_engine'] = engine

break

except Exception as e:

print(f"Error with fallback {engine} engine: {e}")

def \_enhanced\_speak(self, text):

# Implement enhanced voice engine

pass

def \_pyttsx3\_speak(self, text):

engine = pyttsx3.init()

engine.say(text)

engine.runAndWait()

def \_openai\_speak(self, text):

response = openai.Audio.create(

model="tts-1",

voice="alloy",

input=text

)

# Play the audio response

pass

def \_console\_speak(self, text):

print(text)

**2. Advanced AI Model Management**

Add more models and improve the switching mechanism.

python

Copy

# ollama\_manager.py

import ollama

import threading

class OllamaManager:

def \_\_init\_\_(self, config):

self.config = config

self.models = {

'qwen2.5:latest': {'size': '4.7GB', 'specialty': 'General Purpose'},

'llama3.2:latest': {'size': '2.0GB', 'specialty': 'Conversational'},

'hermes3:latest': {'size': '4.7GB', 'specialty': 'Advanced Reasoning'},

'phi-3-mini': {'size': '2.8GB', 'specialty': 'Efficient Processing'},

'qwen3:0.6b': {'size': '522MB', 'specialty': 'Lightweight'},

'mxbai-embed': {'size': '669MB', 'specialty': 'Embeddings'}

}

self.current\_model = self.config.get('llm\_model', 'qwen2.5:latest')

self.lock = threading.Lock()

def switch\_model(self, model\_name):

if model\_name in self.models:

with self.lock:

self.current\_model = model\_name

self.config['llm\_model'] = model\_name

self.\_pull\_model(model\_name)

else:

raise ValueError(f"Model {model\_name} not found")

def \_pull\_model(self, model\_name):

try:

ollama.pull(model\_name)

except Exception as e:

print(f"Error pulling model {model\_name}: {e}")

def get\_model\_info(self):

return self.models.get(self.current\_model, {})

**3. Enhanced GUI Interface**

Add more interactive controls and real-time monitoring.

python

Copy

# gui\_compact.py

import tkinter as tk

from tkinter import ttk, messagebox

import threading

import psutil

import time

class GUI:

def \_\_init\_\_(self, voice\_manager, ollama\_manager, config):

self.voice\_manager = voice\_manager

self.ollama\_manager = ollama\_manager

self.config = config

self.root = tk.Tk()

self.root.title("ULTRON Agent 2.0")

self.\_setup\_ui()

self.\_start\_monitoring()

def \_setup\_ui(self):

# Cyberpunk theme

self.root.configure(bg='#0a0a2a')

style = ttk.Style()

style.configure('TFrame', background='#0a0a2a')

style.configure('TLabel', background='#0a0a2a', foreground='#00ff00')

style.configure('TButton', background='#0a0a2a', foreground='#00ff00')

style.configure('TCombobox', background='#0a0a2a', foreground='#00ff00')

# Voice Testing

voice\_frame = ttk.Frame(self.root)

voice\_frame.pack(pady=10)

ttk.Label(voice\_frame, text="Voice Testing:").pack(side=tk.LEFT)

ttk.Button(voice\_frame, text="Test Voice", command=self.\_test\_voice).pack(side=tk.LEFT, padx=5)

# Model Switching

model\_frame = ttk.Frame(self.root)

model\_frame.pack(pady=10)

ttk.Label(model\_frame, text="Model Switching:").pack(side=tk.LEFT)

self.model\_var = tk.StringVar(value=self.ollama\_manager.current\_model)

model\_dropdown = ttk.Combobox(model\_frame, textvariable=self.model\_var, values=list(self.ollama\_manager.models.keys()))

model\_dropdown.pack(side=tk.LEFT, padx=5)

ttk.Button(model\_frame, text="Switch Model", command=self.\_switch\_model).pack(side=tk.LEFT, padx=5)

# System Monitor

monitor\_frame = ttk.Frame(self.root)

monitor\_frame.pack(pady=10)

ttk.Label(monitor\_frame, text="System Monitor:").pack()

self.cpu\_label = ttk.Label(monitor\_frame, text="CPU: 0%")

self.cpu\_label.pack()

self.memory\_label = ttk.Label(monitor\_frame, text="Memory: 0%")

self.memory\_label.pack()

self.gpu\_label = ttk.Label(monitor\_frame, text="GPU: 0%")

self.gpu\_label.pack()

self.network\_label = ttk.Label(monitor\_frame, text="Network: 0 KB/s")

self.network\_label.pack()

# Screenshot

screenshot\_frame = ttk.Frame(self.root)

screenshot\_frame.pack(pady=10)

ttk.Button(screenshot\_frame, text="Take Screenshot", command=self.\_take\_screenshot).pack()

# File Manager

file\_frame = ttk.Frame(self.root)

file\_frame.pack(pady=10)

ttk.Button(file\_frame, text="Open File Manager", command=self.\_open\_file\_manager).pack()

def \_test\_voice(self):

self.voice\_manager.speak("Testing voice output. This is ULTRON Agent 2.0.")

def \_switch\_model(self):

model\_name = self.model\_var.get()

try:

self.ollama\_manager.switch\_model(model\_name)

messagebox.showinfo("Success", f"Model switched to {model\_name}")

except Exception as e:

messagebox.showerror("Error", f"Failed to switch model: {e}")

def \_take\_screenshot(self):

import pyautogui

screenshot = pyautogui.screenshot()

timestamp = time.strftime("%Y%m%d-%H%M%S")

screenshot.save(f"screenshot\_{timestamp}.png")

messagebox.showinfo("Success", "Screenshot taken and saved.")

def \_open\_file\_manager(self):

import os

os.startfile(os.path.dirname(os.path.abspath(\_\_file\_\_)))

def \_start\_monitoring(self):

def update\_monitor():

while True:

cpu\_percent = psutil.cpu\_percent(interval=1)

memory\_percent = psutil.virtual\_memory().percent

gpu\_percent = self.\_get\_gpu\_usage()

net\_io = psutil.net\_io\_counters()

net\_speed = net\_io.bytes\_sent + net\_io.bytes\_recv

self.cpu\_label.config(text=f"CPU: {cpu\_percent}%")

self.memory\_label.config(text=f"Memory: {memory\_percent}%")

self.gpu\_label.config(text=f"GPU: {gpu\_percent}%")

self.network\_label.config(text=f"Network: {net\_speed / 1024:.2f} KB/s")

time.sleep(1)

threading.Thread(target=update\_monitor, daemon=True).start()

def \_get\_gpu\_usage(self):

try:

import GPUtil

gpus = GPUtil.getGPUs()

if gpus:

return gpus[0].load \* 100

except Exception as e:

print(f"Error getting GPU usage: {e}")

return 0

def run(self):

self.root.mainloop()

**4. Enhanced System Integration**

Add more system monitoring and integration features.

python

Copy

# agent\_core.py

import psutil

import platform

import socket

import threading

import time

class AgentCore:

def \_\_init\_\_(self, config):

self.config = config

self.processes = {}

self.lock = threading.Lock()

def get\_system\_info(self):

info = {

'os': platform.system(),

'os\_version': platform.version(),

'processor': platform.processor(),

'cpu\_cores': psutil.cpu\_count(logical=False),

'total\_memory': psutil.virtual\_memory().total / (1024 \*\* 3), # GB

'gpus': self.\_get\_gpu\_info(),

'network': self.\_get\_network\_info()

}

return info

def \_get\_gpu\_info(self):

try:

import GPUtil

gpus = GPUtil.getGPUs()

return [{'name': gpu.name, 'load': gpu.load \* 100, 'memory': gpu.memoryUtil \* 100} for gpu in gpus]

except Exception as e:

print(f"Error getting GPU info: {e}")

return []

def \_get\_network\_info(self):

info = {

'hostname': socket.gethostname(),

'ip\_address': socket.gethostbyname(socket.gethostname()),

'interfaces': psutil.net\_if\_addrs()

}

return info

def monitor\_processes(self):

def update\_processes():

while True:

with self.lock:

self.processes = {p.pid: p.info for p in psutil.process\_iter(['pid', 'name', 'cpu\_percent', 'memory\_percent'])}

time.sleep(5)

threading.Thread(target=update\_processes, daemon=True).start()

def get\_processes(self):

with self.lock:

return self.processes

**5. Enhanced Startup Script**

Improve the startup script to handle more edge cases.

batch

Copy

@echo off

:: ULTRON Agent 2.0 - Professional Startup Script

:: This script handles system diagnostics, Python verification, Ollama integration, GPU detection, network validation, and process management.

:: Set environment variables

set PYTHON\_VERSION=3.10

set OLLAMA\_URL=https://ollama.ai

set OLLAMA\_MODEL=qwen2.5:latest

:: System Diagnostics

echo 🔍 Performing system diagnostics...

systeminfo > startup.log 2>&1

echo ✅ System diagnostics completed.

:: Python Verification

echo 🐍 Verifying Python installation...

python --version >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Python not found. Please install Python %PYTHON\_VERSION% or higher.

pause

exit /b 1

)

echo ✅ Python verified.

:: Ollama Integration

echo 🤖 Checking Ollama integration...

ollama --version >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Ollama not found. Please install Ollama from %OLLAMA\_URL%.

pause

exit /b 1

)

echo 📥 Pulling Ollama model %OLLAMA\_MODEL%...

ollama pull %OLLAMA\_MODEL% >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Failed to pull Ollama model.

pause

exit /b 1

)

echo ✅ Ollama integration verified.

:: GPU Detection

echo 🖥️ Detecting GPUs...

nvidia-smi >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ⚠️ NVIDIA GPU not found.

) else (

echo ✅ NVIDIA GPU detected.

)

wmic path win32\_VideoController get name >> startup.log 2>&1

echo ✅ GPU detection completed.

:: Network Validation

echo 🌐 Validating network connection...

ping -n 1 google.com >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Network connection failed.

pause

exit /b 1

)

echo ✅ Network connection validated.

:: Process Management

echo 🔧 Managing processes...

tasklist >> startup.log 2>&1

echo ✅ Process management completed.

:: Install Python dependencies

echo 📦 Installing Python dependencies...

pip install -r requirements.txt >> startup.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Failed to install Python dependencies.

pause

exit /b 1

)

echo ✅ Python dependencies installed.

:: Run the agent

echo 🚀 Starting ULTRON Agent 2.0...

python main.py >> ultron.log 2>&1

if %ERRORLEVEL% neq 0 (

echo ❌ Failed to start ULTRON Agent 2.0.

pause

exit /b 1

)

**6. Enhanced Configuration**

Add more configuration options and improve the configuration management.

python

Copy

# ultron\_config.json

{

"openai\_api\_key": "sk-proj-...",

"openai\_organization": "org-...",

"openai\_project": "proj\_...",

"llm\_model": "qwen2.5:latest",

"voice\_engine": "pyttsx3",

"use\_gui": true,

"use\_voice": true,

"system\_monitor": true,

"process\_monitor": true,

"auto\_update": true,

"theme": "cyberpunk",

"language": "en"

}

**7. Enhanced Logging**

Improve the logging system to capture more detailed information.

python

Copy

# logging\_config.py

import logging

import logging.config

import os

def setup\_logging():

log\_dir = "logs"

if not os.path.exists(log\_dir):

os.makedirs(log\_dir)

logging\_config = {

'version': 1,

'formatters': {

'standard': {

'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

},

},

'handlers': {

'console': {

'class': 'logging.StreamHandler',

'formatter': 'standard',

'level': 'INFO'

},

'file': {

'class': 'logging.FileHandler',

'formatter': 'standard',

'filename': os.path.join(log\_dir, 'ultron.log'),

'level': 'DEBUG'

},

'startup': {

'class': 'logging.FileHandler',

'formatter': 'standard',

'filename': os.path.join(log\_dir, 'startup.log'),

'level': 'DEBUG'

},

'error': {

'class': 'logging.FileHandler',

'formatter': 'standard',

'filename': os.path.join(log\_dir, 'error.log'),

'level': 'ERROR'

},

'gui': {

'class': 'logging.FileHandler',

'formatter': 'standard',

'filename': os.path.join(log\_dir, 'ultron\_gui.log'),

'level': 'DEBUG'

}

},

'loggers': {

'': {

'handlers': ['console', 'file'],

'level': 'DEBUG',

'propagate': False

},

'startup': {

'handlers': ['console', 'startup'],

'level': 'DEBUG',

'propagate': False

},

'error': {

'handlers': ['console', 'error'],

'level': 'ERROR',

'propagate': False

},

'gui': {

'handlers': ['console', 'gui'],

'level': 'DEBUG',

'propagate': False

}

}

}

logging.config.dictConfig(logging\_config)

**8. Enhanced API Access**

Add more API endpoints and improve the API access.

python

Copy

# api.py

from flask import Flask, jsonify, request

import threading

class API:

def \_\_init\_\_(self, voice\_manager, ollama\_manager, agent\_core, config):

self.voice\_manager = voice\_manager

self.ollama\_manager = ollama\_manager

self.agent\_core = agent\_core

self.config = config

self.app = Flask(\_\_name\_\_)

self.\_setup\_routes()

def \_setup\_routes(self):

@self.app.route('/api/status', methods=['GET'])

def get\_status():

status = {

'system\_info': self.agent\_core.get\_system\_info(),

'processes': self.agent\_core.get\_processes(),

'model\_info': self.ollama\_manager.get\_model\_info(),

'voice\_engine': self.config.get('voice\_engine', 'pyttsx3')

}

return jsonify(status)

@self.app.route('/api/speak', methods=['POST'])

def speak():

data = request.json

text = data.get('text', '')

if text:

self.voice\_manager.speak(text)

return jsonify({'status': 'success'})

else:

return jsonify({'status': 'error', 'message': 'No text provided'}), 400

@self.app.route('/api/switch\_model', methods=['POST'])

def switch\_model():

data = request.json

model\_name = data.get('model\_name', '')

if model\_name:

try:

self.ollama\_manager.switch\_model(model\_name)

return jsonify({'status': 'success'})

except Exception as e:

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

else:

return jsonify({'status': 'error', 'message': 'No model name provided'}), 400

@self.app.route('/api/system\_info', methods=['GET'])

def get\_system\_info():

return jsonify(self.agent\_core.get\_system\_info())

@self.app.route('/api/processes', methods=['GET'])

def get\_processes():

return jsonify(self.agent\_core.get\_processes())

def run(self):

threading.Thread(target=lambda: self.app.run(host='0.0.0.0', port=5000), daemon=True).start()

**9. Enhanced Main Script**

Improve the main script to handle more edge cases and provide better integration.

python

Copy

# main.py

import json

import logging

from voice\_manager import VoiceManager

from ollama\_manager import OllamaManager

from gui\_compact import GUI

from agent\_core import AgentCore

from api import API

from logging\_config import setup\_logging

def load\_config():

try:

with open('ultron\_config.json', 'r') as f:

config = json.load(f)

return config

except Exception as e:

logging.error(f"Error loading config: {e}")

return {}

def save\_config(config):

try:

with open('ultron\_config.json', 'w') as f:

json.dump(config, f, indent=4)

except Exception as e:

logging.error(f"Error saving config: {e}")

def main():

setup\_logging()

logging.info("Starting ULTRON Agent 2.0")

config = load\_config()

voice\_manager = VoiceManager(config)

ollama\_manager = OllamaManager(config)

agent\_core = AgentCore(config)

if config.get('system\_monitor', True):

agent\_core.monitor\_processes()

if config.get('use\_gui', True):

gui = GUI(voice\_manager, ollama\_manager, config)

gui\_thread = threading.Thread(target=gui.run, daemon=True)

gui\_thread.start()

api = API(voice\_manager, ollama\_manager, agent\_core, config)

api.run()

logging.info("ULTRON Agent 2.0 started successfully")

try:

while True:

time.sleep(1)

except KeyboardInterrupt:

logging.info("Shutting down ULTRON Agent 2.0")

save\_config(config)

if \_\_name\_\_ == "\_\_main\_\_":

main()

These enhancements should significantly improve the functionality and robustness of ULTRON Agent 2.0. Each component is designed to work together seamlessly, providing a comprehensive AI assistant experience.

explain this scripts function and usefulness to me

<script>

#!/usr/bin/env python3

"""

AI-Powered Content Generator - Automated Project Setup and Model Integration Script

This script automates the process of setting up a complete AI project with local model integration,

GitHub repository creation, and automated content generation capabilities.

Author: AI Project Bot

Date: 2023-06-01

Version: 1.0.0

"""

import os

import sys

import json

import yaml

import logging

import subprocess

import shutil

from pathlib import Path

from datetime import datetime

from typing import Dict, List, Any, Optional

import tempfile

# Third-party imports with error handling

try:

import torch

from transformers import AutoTokenizer, AutoModel, AutoModelForCausalLM, pipeline

import github3

import requests

except ImportError as e:

print(f"Error importing required packages: {e}")

print("Please install required packages: pip install torch transformers github3.py pyyaml requests")

sys.exit(1)

# Configure logging

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler('project\_automation.log'),

logging.StreamHandler(sys.stdout)

]

)

logger = logging.getLogger(\_\_name\_\_)

class ProjectAutomationError(Exception):

"""Custom exception for project automation errors"""

pass

class LocalModelManager:

"""

Manages local AI models including loading, configuration, and text generation

"""

def \_\_init\_\_(self, models\_config: List[Dict[str, Any]]):

"""

Initialize the LocalModelManager with model configurations

Args:

models\_config: List of dictionaries containing model configurations

"""

self.models\_config = models\_config

self.loaded\_models = {}

self.tokenizers = {}

def setup\_models(self) -> bool:

"""

Set up and validate all configured local models

Returns:

bool: True if all models are set up successfully, False otherwise

"""

logger.info("Setting up local models...")

for model\_config in self.models\_config:

try:

model\_name = model\_config['name']

model\_path = model\_config['path']

model\_type = model\_config['type']

logger.info(f"Setting up model: {model\_name}")

# Check if model directory exists

if not os.path.exists(model\_path):

logger.warning(f"Model path not found: {model\_path}. Attempting to download...")

self.\_download\_model(model\_config)

# Load tokenizer

tokenizer = AutoTokenizer.from\_pretrained(

model\_config['tokenizer'],

local\_files\_only=False

)

self.tokenizers[model\_name] = tokenizer

# Load model based on type

if model\_type == 'text-generation':

model = AutoModelForCausalLM.from\_pretrained(

model\_path if os.path.exists(model\_path) else model\_config['tokenizer'],

local\_files\_only=False

)

else:

model = AutoModel.from\_pretrained(

model\_path if os.path.exists(model\_path) else model\_config['tokenizer'],

local\_files\_only=False

)

self.loaded\_models[model\_name] = model

logger.info(f"Successfully loaded model: {model\_name}")

except Exception as e:

logger.error(f"Failed to setup model {model\_config['name']}: {str(e)}")

return False

return True

def \_download\_model(self, model\_config: Dict[str, Any]) -> None:

"""

Download model if not available locally

Args:

model\_config: Dictionary containing model configuration

"""

model\_name = model\_config['name']

model\_path = model\_config['path']

tokenizer\_name = model\_config['tokenizer']

try:

# Create directory if it doesn't exist

os.makedirs(model\_path, exist\_ok=True)

# Download tokenizer and model

logger.info(f"Downloading tokenizer for {model\_name}...")

tokenizer = AutoTokenizer.from\_pretrained(tokenizer\_name)

tokenizer.save\_pretrained(model\_path)

logger.info(f"Downloading model {model\_name}...")

if model\_config['type'] == 'text-generation':

model = AutoModelForCausalLM.from\_pretrained(tokenizer\_name)

else:

model = AutoModel.from\_pretrained(tokenizer\_name)

model.save\_pretrained(model\_path)

logger.info(f"Successfully downloaded model: {model\_name}")

except Exception as e:

logger.error(f"Failed to download model {model\_name}: {str(e)}")

raise ProjectAutomationError(f"Model download failed: {str(e)}")

def generate\_content(self, model\_name: str, prompt: str, max\_length: int = 100) -> str:

"""

Generate content using the specified model

Args:

model\_name: Name of the model to use

prompt: Input prompt for content generation

max\_length: Maximum length of generated content

Returns:

str: Generated content

"""

try:

if model\_name not in self.loaded\_models:

raise ProjectAutomationError(f"Model {model\_name} not loaded")

model = self.loaded\_models[model\_name]

tokenizer = self.tokenizers[model\_name]

# Create pipeline for text generation

generator = pipeline(

'text-generation',

model=model,

tokenizer=tokenizer,

max\_length=max\_length,

do\_sample=True,

temperature=0.7

)

result = generator(prompt)

return result[0]['generated\_text']

except Exception as e:

logger.error(f"Content generation failed: {str(e)}")

return f"Error generating content: {str(e)}"

class GitHubManager:

"""

Manages GitHub repository operations including creation, file management, and commits

"""

def \_\_init\_\_(self, credentials: Dict[str, str]):

"""

Initialize GitHubManager with credentials

Args:

credentials: Dictionary containing GitHub credentials and repository info

"""

self.credentials = credentials

self.github\_client = None

self.repository = None

def authenticate(self) -> bool:

"""

Authenticate with GitHub using access token

Returns:

bool: True if authentication successful, False otherwise

"""

try:

self.github\_client = github3.login(token=self.credentials['access\_token'])

# Test authentication by getting user info

user = self.github\_client.me()

logger.info(f"Successfully authenticated as: {user.login}")

return True

except Exception as e:

logger.error(f"GitHub authentication failed: {str(e)}")

return False

def create\_repository(self, project\_requirements: Dict[str, Any]) -> bool:

"""

Create a new GitHub repository with project requirements

Args:

project\_requirements: Dictionary containing project specifications

Returns:

bool: True if repository created successfully, False otherwise

"""

try:

repo\_name = self.credentials['repo\_name']

description = project\_requirements['description']

logger.info(f"Creating repository: {repo\_name}")

# Check if repository already exists

try:

existing\_repo = self.github\_client.repository(

self.credentials['repo\_owner'],

repo\_name

)

if existing\_repo:

logger.info(f"Repository {repo\_name} already exists")

self.repository = existing\_repo

return True

except:

pass # Repository doesn't exist, continue with creation

# Create new repository

self.repository = self.github\_client.create\_repository(

name=repo\_name,

description=description,

private=False,

auto\_init=True,

gitignore\_template='Python'

)

logger.info(f"Successfully created repository: {repo\_name}")

return True

except Exception as e:

logger.error(f"Failed to create repository: {str(e)}")

return False

def create\_file(self, path: str, content: str, commit\_message: str) -> bool:

"""

Create a file in the repository

Args:

path: File path in repository

content: File content

commit\_message: Commit message

Returns:

bool: True if file created successfully, False otherwise

"""

try:

# Check if file already exists

try:

existing\_file = self.repository.file\_contents(path)

if existing\_file:

# Update existing file

existing\_file.update(

commit\_message,

content.encode('utf-8'),

sha=existing\_file.sha

)

logger.info(f"Updated existing file: {path}")

return True

except:

pass # File doesn't exist, create new

# Create new file

self.repository.create\_file(

path=path,

message=commit\_message,

content=content.encode('utf-8')

)

logger.info(f"Created file: {path}")

return True

except Exception as e:

logger.error(f"Failed to create file {path}: {str(e)}")

return False

def create\_directory\_structure(self, structure: Dict[str, List[str]]) -> bool:

"""

Create directory structure with files in the repository

Args:

structure: Dictionary mapping directories to file lists

Returns:

bool: True if structure created successfully, False otherwise

"""

try:

for directory, files in structure.items():

for file\_name in files:

file\_path = f"{directory}/{file\_name}" if directory else file\_name

# Generate appropriate content based on file type

content = self.\_generate\_file\_content(file\_name, directory)

success = self.create\_file(

file\_path,

content,

f"Add {file\_name}"

)

if not success:

logger.error(f"Failed to create {file\_path}")

return False

logger.info("Successfully created directory structure")

return True

except Exception as e:

logger.error(f"Failed to create directory structure: {str(e)}")

return False

def \_generate\_file\_content(self, file\_name: str, directory: str) -> str:

"""

Generate appropriate content for different file types

Args:

file\_name: Name of the file

directory: Directory containing the file

Returns:

str: Generated file content

"""

if file\_name == "README.md":

return self.\_generate\_readme\_content()

elif file\_name == "main.py":

return self.\_generate\_main\_py\_content()

elif file\_name == "model\_handler.py":

return self.\_generate\_model\_handler\_content()

elif file\_name == "github\_manager.py":

return self.\_generate\_github\_manager\_content()

elif file\_name == "content\_generator.py":

return self.\_generate\_content\_generator\_content()

elif file\_name.endswith(".yaml"):

return self.\_generate\_yaml\_config\_content(file\_name)

elif file\_name.startswith("test\_"):

return self.\_generate\_test\_content(file\_name)

elif file\_name == "CONTRIBUTING.md":

return self.\_generate\_contributing\_content()

elif file\_name == "LICENSE":

return self.\_generate\_license\_content()

else:

return f"# {file\_name}\n\n# TODO: Implement {file\_name}\n"

def \_generate\_readme\_content(self) -> str:

"""Generate README.md content"""

return """# AI-Powered Content Generator

An automated content generation system using local AI models to create blog posts, social media updates, and product descriptions.

## Features

- Integration with multiple local AI models for text generation

- GitHub repository creation and management

- Automated content categorization and tagging

- Output in multiple formats (Markdown, HTML, plain text)

- Version control and change tracking

## Requirements

- Python 3.8+

- PyTorch

- Transformers

- GitHub3.py

- PyYAML

## Installation

```bash

pip install -r requirements.txt

```

## Usage

```bash

python src/main.py

```

## Configuration

Configure your models and GitHub settings in the config files:

- `config/models\_config.yaml` - Local model configurations

- `config/github\_config.yaml` - GitHub API settings

## License

MIT License - see LICENSE file for details

"""

def \_generate\_main\_py\_content(self) -> str:

"""Generate main.py content"""

return '''#!/usr/bin/env python3

"""

AI-Powered Content Generator - Main Application Entry Point

"""

import logging

import yaml

from pathlib import Path

from model\_handler import LocalModelManager

from github\_manager import GitHubManager

from content\_generator import ContentGenerator

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

def load\_config(config\_path: str) -> dict:

"""Load configuration from YAML file"""

try:

with open(config\_path, 'r') as file:

return yaml.safe\_load(file)

except Exception as e:

logger.error(f"Failed to load config {config\_path}: {e}")

return {}

def main():

"""Main application entry point"""

logger.info("Starting AI-Powered Content Generator...")

# Load configurations

models\_config = load\_config('config/models\_config.yaml')

github\_config = load\_config('config/github\_config.yaml')

# Initialize components

model\_manager = LocalModelManager(models\_config.get('models', []))

github\_manager = GitHubManager(github\_config)

content\_generator = ContentGenerator(model\_manager, github\_manager)

# Setup models

if not model\_manager.setup\_models():

logger.error("Failed to setup models")

return 1

# Authenticate with GitHub

if not github\_manager.authenticate():

logger.error("Failed to authenticate with GitHub")

return 1

# Generate content

content\_generator.run()

logger.info("Content generation completed successfully")

return 0

if \_\_name\_\_ == "\_\_main\_\_":

exit(main())

'''

def \_generate\_model\_handler\_content(self) -> str:

"""Generate model\_handler.py content"""

return '''"""

Local Model Handler for AI Content Generation

"""

import logging

import torch

from transformers import AutoTokenizer, AutoModel, AutoModelForCausalLM, pipeline

from typing import Dict, List, Any

logger = logging.getLogger(\_\_name\_\_)

class LocalModelManager:

"""Manages local AI models for content generation"""

def \_\_init\_\_(self, models\_config: List[Dict[str, Any]]):

self.models\_config = models\_config

self.loaded\_models = {}

self.tokenizers = {}

def setup\_models(self) -> bool:

"""Setup and load all configured models"""

# Implementation here

pass

def generate\_content(self, model\_name: str, prompt: str) -> str:

"""Generate content using specified model"""

# Implementation here

pass

'''

def \_generate\_github\_manager\_content(self) -> str:

"""Generate github\_manager.py content"""

return '''"""

GitHub Repository Management

"""

import logging

import github3

from typing import Dict, Any

logger = logging.getLogger(\_\_name\_\_)

class GitHubManager:

"""Manages GitHub repository operations"""

def \_\_init\_\_(self, credentials: Dict[str, str]):

self.credentials = credentials

self.github\_client = None

self.repository = None

def authenticate(self) -> bool:

"""Authenticate with GitHub"""

# Implementation here

pass

def create\_repository(self, project\_requirements: Dict[str, Any]) -> bool:

"""Create new repository"""

# Implementation here

pass

'''

def \_generate\_content\_generator\_content(self) -> str:

"""Generate content\_generator.py content"""

return '''"""

Content Generation Orchestrator

"""

import logging

from typing import Dict, Any

logger = logging.getLogger(\_\_name\_\_)

class ContentGenerator:

"""Orchestrates content generation process"""

def \_\_init\_\_(self, model\_manager, github\_manager):

self.model\_manager = model\_manager

self.github\_manager = github\_manager

def run(self):

"""Main content generation workflow"""

# Implementation here

pass

'''

def \_generate\_yaml\_config\_content(self, file\_name: str) -> str:

"""Generate YAML configuration content"""

if "models" in file\_name:

return '''# Local Models Configuration

models:

- name: "GPT-2-small"

type: "text-generation"

path: "C:\\\\AI\_Models\\\\gpt2-small"

framework: "pytorch"

tokenizer: "gpt2"

max\_length: 1024

temperature: 0.7

'''

else:

return '''# GitHub Configuration

access\_token: "your\_token\_here"

username: "your\_username"

email: "your\_email@example.com"

repo\_owner: "your\_org"

repo\_name: "your\_repo"

'''

def \_generate\_test\_content(self, file\_name: str) -> str:

"""Generate test file content"""

class\_name = file\_name.replace("test\_", "").replace(".py", "").title().replace("\_", "")

return f'''"""

Unit tests for {class\_name}

"""

import unittest

from unittest.mock import Mock, patch

class Test{class\_name}(unittest.TestCase):

"""Test cases for {class\_name}"""

def setUp(self):

"""Set up test fixtures"""

pass

def test\_placeholder(self):

"""Placeholder test"""

self.assertTrue(True)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

'''

def \_generate\_contributing\_content(self) -> str:

"""Generate CONTRIBUTING.md content"""

return '''# Contributing to AI-Powered Content Generator

Thank you for your interest in contributing! Please follow these guidelines:

## Development Setup

1. Fork the repository

2. Clone your fork

3. Create a virtual environment

4. Install dependencies: `pip install -r requirements.txt`

## Making Changes

1. Create a feature branch

2. Make your changes

3. Add tests for new functionality

4. Run tests: `python -m pytest`

5. Submit a pull request

## Code Style

- Follow PEP 8

- Add docstrings to all functions

- Include type hints where appropriate

'''

def \_generate\_license\_content(self) -> str:

"""Generate LICENSE content"""

return '''MIT License

Copyright (c) 2023 AI Project Bot

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

'''

class ProjectGenerator:

"""

Main class for orchestrating the entire project automation process

"""

def \_\_init\_\_(self, project\_requirements: Dict[str, Any],

local\_models: List[Dict[str, Any]],

github\_credentials: Dict[str, str]):

"""

Initialize ProjectGenerator with all necessary configurations

Args:

project\_requirements: Project specification dictionary

local\_models: List of local model configurations

github\_credentials: GitHub authentication credentials

"""

self.project\_requirements = project\_requirements

self.local\_models = local\_models

self.github\_credentials = github\_credentials

# Initialize managers

self.model\_manager = LocalModelManager(local\_models)

self.github\_manager = GitHubManager(github\_credentials)

def run\_automation(self) -> bool:

"""

Execute the complete project automation workflow

Returns:

bool: True if automation completed successfully, False otherwise

"""

try:

logger.info("Starting project automation workflow...")

# Step 1: Setup local models

logger.info("Step 1: Setting up local models...")

if not self.model\_manager.setup\_models():

raise ProjectAutomationError("Failed to setup local models")

# Step 2: Authenticate with GitHub

logger.info("Step 2: Authenticating with GitHub...")

if not self.github\_manager.authenticate():

raise ProjectAutomationError("Failed to authenticate with GitHub")

# Step 3: Create GitHub repository

logger.info("Step 3: Creating GitHub repository...")

if not self.github\_manager.create\_repository(self.project\_requirements):

raise ProjectAutomationError("Failed to create GitHub repository")

# Step 4: Generate project structure

logger.info("Step 4: Creating project structure...")

if not self.github\_manager.create\_directory\_structure(

self.project\_requirements['project\_structure']

):

raise ProjectAutomationError("Failed to create project structure")

# Step 5: Generate and commit additional files

logger.info("Step 5: Generating additional project files...")

self.\_generate\_additional\_files()

# Step 6: Test model integration

logger.info("Step 6: Testing model integration...")

self.\_test\_model\_integration()

logger.info("Project automation completed successfully!")

return True

except Exception as e:

logger.error(f"Project automation failed: {str(e)}")

return False

def \_generate\_additional\_files(self) -> None:

"""Generate additional project files like requirements.txt, setup.py, etc."""

# Generate requirements.txt

requirements\_content = """torch>=1.9.0

transformers>=4.20.0

github3.py>=3.2.0

pyyaml>=6.0

requests>=2.28.0

pytest>=7.0.0

"""

self.github\_manager.create\_file(

"requirements.txt",

requirements\_content,

"Add requirements.txt"

)

# Generate setup.py

setup\_content = f'''from setuptools import setup, find\_packages

setup(

name="{self.project\_requirements['project\_name'].lower().replace(' ', '-')}",

version="1.0.0",

description="{self.project\_requirements['description']}",

packages=find\_packages(),

install\_requires=[

"torch>=1.9.0",

"transformers>=4.20.0",

"github3.py>=3.2.0",

"pyyaml>=6.0",

"requests>=2.28.0",

],

python\_requires=">=3.8",

)

'''

self.github\_manager.create\_file(

"setup.py",

setup\_content,

"Add setup.py"

)

# Generate .gitignore

gitignore\_content = """# Python

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

\*.so

.Python

build/

develop-eggs/

dist/

downloads/

eggs/

.eggs/

lib/

lib64/

parts/

sdist/

var/

wheels/

\*.egg-info/

.installed.cfg

\*.egg

# Virtual Environment

venv/

env/

ENV/

# IDE

.vscode/

.idea/

\*.swp

\*.swo

# OS

.DS\_Store

Thumbs.db

# Logs

\*.log

# Models (if too large)

models/

\*.bin

\*.safetensors

# Config files with secrets

config/\*\_secret.yaml

"""

self.github\_manager.create\_file(

".gitignore",

gitignore\_content,

"Add .gitignore"

)

def \_test\_model\_integration(self) -> None:

"""Test integration with local models by generating sample content"""

test\_prompts = [

"Write a short blog post about artificial intelligence:",

"Create a product description for a smart home device:",

"Generate a social media post about sustainable technology:"

]

results = []

for i, prompt in enumerate(test\_prompts):

try:

# Use the first available text-generation model

text\_gen\_models = [m for m in self.local\_models if m['type'] == 'text-generation']

if text\_gen\_models:

model\_name = text\_gen\_models[0]['name']

generated\_content = self.model\_manager.generate\_content(

model\_name,

prompt,

max\_length=200

)

results.append(f"## Test {i+1}\n\*\*Prompt:\*\* {prompt}\n\*\*Generated:\*\*\n{generated\_content}\n")

except Exception as e:

results.append(f"## Test {i+1}\n\*\*Prompt:\*\* {prompt}\n\*\*Error:\*\* {str(e)}\n")

# Create test results file

test\_results\_content = f"""# Model Integration Test Results

Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

{''.join(results)}

## Model Configuration Used

{json.dumps(self.local\_models, indent=2)}

"""

self.github\_manager.create\_file(

"docs/test\_results.md",

test\_results\_content,

"Add model integration test results"

)

def main():

"""

Main execution function - orchestrates the entire automation process

"""

# Project Requirements Configuration

project\_requirements = {

"project\_name": "AI-Powered Content Generator",

"description": "Develop an automated content generation system using local AI models to create blog posts, social media updates, and product descriptions.",

"features": [

"Integration with multiple local AI models for text generation",

"GitHub repository creation and management",

"Automated content categorization and tagging",

"Output in multiple formats (Markdown, HTML, plain text)",

"Version control and change tracking"

],

"technical\_requirements": {

"language": "Python 3.8+",

"dependencies": [

"transformers",

"pytorch",

"github3.py",

"pyyaml"

],

"input\_formats": ["CSV", "JSON"],

"output\_formats": ["Markdown", "HTML", "TXT"]

},

"project\_structure": {

"src": ["main.py", "model\_handler.py", "github\_manager.py", "content\_generator.py"],

"tests": ["test\_model\_handler.py", "test\_github\_manager.py", "test\_content\_generator.py"],

"docs": ["README.md", "CONTRIBUTING.md", "LICENSE"],

"config": ["models\_config.yaml", "github\_config.yaml"]

},

"timeline": {

"start\_date": "2023-06-01",

"end\_date": "2023-07-15",

"milestones": [

{"name": "Project Setup", "date": "2023-06-05"},

{"name": "Model Integration", "date": "2023-06-20"},

{"name": "GitHub Automation", "date": "2023-07-01"},

{"name": "Testing and Documentation", "date": "2023-07-10"}

]

}

}

# Local Models Configuration

local\_models = [

{

"name": "GPT-2-small",

"type": "text-generation",

"path": "C:\\AI\_Models\\gpt2-small",

"framework": "pytorch",

"tokenizer": "gpt2",

"max\_length": 1024,

"temperature": 0.7

},

{

"name": "BERT-base-uncased",

"type": "text-classification",

"path": "C:\\AI\_Models\\bert-base-uncased",

"framework": "pytorch",

"tokenizer": "bert-base-uncased",

"max\_length": 512

},

{

"name": "T5-small",

"type": "text-to-text",

"path": "C:\\AI\_Models\\t5-small",

"framework": "pytorch",

"tokenizer": "t5-small",

"max\_length": 512

}

]

# GitHub Credentials Configuration

# IMPORTANT: Replace with your actual credentials

github\_credentials = {

"access\_token": "ghp\_1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6p7q8r9", # Replace with your token

"username": "ai\_project\_bot", # Replace with your username

"email": "ai\_project\_bot@example.com", # Replace with your email

"repo\_owner": "TechInnovators", # Replace with your organization/username

"repo\_name": "AI-Powered-Content-Generator" # Repository name

}

try:

# Initialize and run project automation

project\_generator = ProjectGenerator(

project\_requirements=project\_requirements,

local\_models=local\_models,

github\_credentials=github\_credentials

)

success = project\_generator.run\_automation()

if success:

logger.info("=" \* 60)

logger.info("PROJECT AUTOMATION COMPLETED SUCCESSFULLY!")

logger.info("=" \* 60)

logger.info(f"Repository created: https://github.com/{github\_credentials['repo\_owner']}/{github\_credentials['repo\_name']}")

logger.info("Check the repository for all generated files and documentation.")

return 0

else:

logger.error("Project automation failed. Check logs for details.")

return 1

except KeyboardInterrupt:

logger.info("Project automation interrupted by user.")

return 1

except Exception as e:

logger.error(f"Unexpected error during project automation: {str(e)}")

return 1

if \_\_name\_\_ == "\_\_main\_\_":

"""

Script execution entry point

To run this script:

1. Install required dependencies:

pip install torch transformers github3.py pyyaml requests

2. Update the GitHub credentials in the main() function with your actual:

- GitHub access token (with repo permissions)

- Username and email

- Repository owner and name

3. Verify local model paths exist or will be created

4. Run the script:

python project\_automation\_script.py

The script will:

- Set up and validate local AI models

- Create a GitHub repository with complete project structure

- Generate all necessary project files

- Test model integration

- Provide a complete, ready-to-use AI content generation project

"""

# Set up error handling for Windows-specific issues

if sys.platform.startswith('win'):

# Enable ANSI color codes on Windows

os.system('color')

# Set UTF-8 encoding for Windows console

if hasattr(sys.stdout, 'reconfigure'):

sys.stdout.reconfigure(encoding='utf-8')

# Execute main automation workflow

exit\_code = main()

# Provide final status message

if exit\_code == 0:

print("\n" + "="\*60)

print("SUCCESS: Project automation completed successfully!")

print("Check your GitHub repository for the generated project.")

print("="\*60)

else:

print("\n" + "="\*60)

print("FAILED: Project automation encountered errors.")

print("Check the log file 'project\_automation.log' for details.")

print("="\*60)

sys.exit(exit\_code)

</script>

can you modify it to run on my system as an add on for the ultron agent project.

can you put my models in the script please see below

NAME ID SIZE MODIFIED

qwen2.5vl:latest 5ced39dfa4ba 6.0 GB 2 hours ago

qwen3:0.6b 7df6b6e09427 522 MB 8 days ago

qikfox/Eleven:latest a80c4f17acd5 2.0 GB 12 days ago

llama3.2:latest a80c4f17acd5 2.0 GB 12 days ago

qwen2.5:latest 845dbda0ea48 4.7 GB 13 days ago

mxbai-embed-large:latest 468836162de7 669 MB 2 weeks ago

Qwen2.5-7B-Mini.Q5\_K\_S:latest 52c6d8efdc1b 1.9 GB 3 weeks ago

L3.2-8X3B-MOE-Dark-Champion-Inst-18.4B-uncen-ablit\_D\_AU-Q3\_k\_s:latest f741da88c5b7 8.3 GB 3 weeks ago

phi-3-mini-128k-instruct.Q5\_K\_M:latest 5a7f79393205 2.8 GB 3 weeks ago

hermes3:latest 4f6b83f30b62 4.7 GB 4 weeks ago

hermes3:8b 4f6b83f30b62 4.7 GB 4 weeks ago

**ultron\_addons/\_\_init\_\_.py**

python

CopyEdit

"""

ultron\_addons

================

This package contains extensions and add‑ons for the Ultron Agent project. The goal

is to provide modular components that can be dropped into the existing Ultron

architecture without modifying the core repository. These add‑ons include

connectors for new services, enhanced memory backends, advanced brain

implementations, additional tool definitions, and utilities to manage

evolution of the agent over time.

All modules are designed to be self‑contained and have minimal dependencies

outside of the Python standard library. When necessary, optional third‑party

libraries can be used; however, the classes will degrade gracefully if those

libraries are unavailable. See the documentation in each module for details.

"""

from .connectors import Connector, GoogleDriveConnector

from .memory\_enhanced import VectorMemory

from .brain\_extension import AdvancedUltronBrain

from .tools import SearchTool, CalculatorTool

from .evolution import EvolutionManager

\_\_all\_\_ = [

"Connector",

"GoogleDriveConnector",

"VectorMemory",

"AdvancedUltronBrain",

"SearchTool",

"CalculatorTool",

"EvolutionManager",

]

**ultron\_addons/connectors.py**

python

CopyEdit

"""

connectors.py

==============

This module defines a simple connector interface and example implementations

for integrating the Ultron Agent with external systems. A connector is

responsible for authenticating with a third‑party service and exposing a

uniform set of methods for retrieving or storing data. Connectors can be

registered with the agent and used by tools or memory backends to access

external resources.

The base `Connector` class defines the minimal API for all connectors.

Specific connectors such as `GoogleDriveConnector` extend this interface

and provide concrete implementations. Additional connectors for services

like Slack, Gmail, or custom APIs can be added by following the same

pattern.

"""

from \_\_future\_\_ import annotations

from typing import Any, Dict, List, Optional

import logging

class Connector:

"""Abstract base class for all connectors.

Subclasses should implement the `connect`, `fetch`, and

`store` methods. These methods provide a high‑level API for

interacting with an external service. If a particular operation is

unsupported, subclasses may raise ``NotImplementedError``.

"""

name: str = "BaseConnector"

description: str = "Base class for all connectors."

def \_\_init\_\_(self, config: Optional[Dict[str, Any]] = None) -> None:

self.config = config or {}

self.connected = False

def connect(self) -> None:

"""Establish a connection to the external service.

Concrete connectors should override this method to perform any

authentication or setup required. For example, a Google Drive

connector might load OAuth credentials here. By default this

method simply marks the connector as connected.

"""

logging.debug(f"[{self.name}] Connecting using config: {self.config}")

self.connected = True

def fetch(self, query: str) -> List[Any]:

"""Fetch data from the external service matching the given query.

:param query: An arbitrary search string or identifier used to

retrieve resources.

:returns: A list of results.

:raises NotImplementedError: if the connector does not support

fetching data.

"""

raise NotImplementedError(f"{self.name} does not implement fetch()")

def store(self, data: Any, destination: Optional[str] = None) -> str:

"""Store data to the external service.

:param data: Arbitrary data to upload or save.

:param destination: Optional location within the service where the

data should be stored.

:returns: A string identifier for the stored resource (e.g. file ID).

:raises NotImplementedError: if the connector does not support

storing data.

"""

raise NotImplementedError(f"{self.name} does not implement store()")

class GoogleDriveConnector(Connector):

"""Example connector for Google Drive.

This connector uses the Google Drive API to list files, download

documents, and upload new content. Authentication is performed lazily

on first use; if the ``googleapiclient`` library is unavailable, the

connector will operate in a degraded mode and simply log requests.

"""

name = "GoogleDriveConnector"

description = "Connector for accessing Google Drive files and folders."

def \_\_init\_\_(self, credentials\_path: Optional[str] = None, \*\*kwargs: Any) -> None:

super().\_\_init\_\_(kwargs)

self.credentials\_path = credentials\_path

self.service = None

def connect(self) -> None:

# Attempt to authenticate with Google Drive API. We avoid importing

# googleapiclient at module import time to keep dependencies

# optional. Users can install google-api-python-client if needed.

logging.debug("[GoogleDriveConnector] Attempting to connect to Google Drive")

try:

from google.oauth2 import service\_account

from googleapiclient.discovery import build

if not self.credentials\_path:

raise ValueError("Missing credentials\_path for Google Drive connection")

creds = service\_account.Credentials.from\_service\_account\_file(

self.credentials\_path,

scopes=[

"https://www.googleapis.com/auth/drive.readonly",

"https://www.googleapis.com/auth/drive.file",

],

)

self.service = build('drive', 'v3', credentials=creds)

self.connected = True

logging.info("[GoogleDriveConnector] Connected to Google Drive API")

except Exception as e:

# Fallback: mark as connected but in offline mode

logging.warning(

f"[GoogleDriveConnector] Could not connect to Google Drive: {e}. Operating in offline mode."

)

self.connected = False

def fetch(self, query: str) -> List[Dict[str, Any]]:

"""Search for files in Google Drive matching the query.

If the API is unavailable, return an empty list and log the query.

"""

logging.debug(f"[GoogleDriveConnector] Fetch called with query: {query}")

if self.service:

try:

results = self.service.files().list(q=f"name contains '{query}'", pageSize=10).execute()

return results.get('files', [])

except Exception as e:

logging.error(f"[GoogleDriveConnector] Error fetching files: {e}")

# Offline or error: return empty list

logging.info(f"[GoogleDriveConnector] Operating offline. Returning no results for query: {query}")

return []

def store(self, data: Any, destination: Optional[str] = None) -> str:

"""Upload a file to Google Drive."""

logging.debug(f"[GoogleDriveConnector] Store called with data: {data}, destination: {destination}")

if self.service and isinstance(data, str):

try:

from googleapiclient.http import MediaFileUpload

file\_metadata = {'name': data.split('/')[-1]}

if destination:

file\_metadata['parents'] = [destination]

media = MediaFileUpload(data, resumable=True)

file = self.service.files().create(

body=file\_metadata,

media\_body=media,

fields='id'

).execute()

file\_id = file.get('id', '')

logging.info(f"[GoogleDriveConnector] Uploaded file, id: {file\_id}")

return file\_id

except Exception as e:

logging.error(f"[GoogleDriveConnector] Error uploading file: {e}")

# Offline or unsupported data type

logging.info(f"[GoogleDriveConnector] Operating offline. Cannot upload {data}")

return ''

**ultron\_addons/memory\_enhanced.py**

python

CopyEdit

"""

memory\_enhanced.py

===================

Extended memory implementations for the Ultron Agent. These classes build

upon the basic Memory class provided in the core repository (see

`memory.py`) and add features such as vector embeddings and persistent

storage via external databases.

"""

from \_\_future\_\_ import annotations

import logging

from typing import Any, Dict, List

try:

# Optional import for embedding; if unavailable, we will use simple text

# hashing instead.

from sentence\_transformers import SentenceTransformer

import numpy as np

except ImportError: # pragma: no cover

SentenceTransformer = None # type: ignore

np = None # type: ignore

from collections import deque

class VectorMemory:

"""An enhanced memory that stores both raw items and vector embeddings."""

def \_\_init\_\_(self, short\_term\_limit: int = 10, model\_name: str = 'all-MiniLM-L6-v2') -> None:

self.short\_term\_memory: deque[Any] = deque(maxlen=short\_term\_limit)

self.long\_term\_memory: Dict[str, Dict[str, Any]] = {}

self.model\_name = model\_name

if SentenceTransformer:

try:

self.embedder = SentenceTransformer(model\_name)

logging.info(f"[VectorMemory] Loaded embedding model: {model\_name}")

except Exception as e:

logging.error(f"[VectorMemory] Failed to load embedding model: {e}")

self.embedder = None

else:

self.embedder = None

def \_embed(self, text: str) -> List[float]:

"""Compute a vector representation for the given text."""

if self.embedder:

try:

vec = self.embedder.encode([text])[0]

return vec.tolist() # type: ignore[no-any-return]

except Exception as e:

logging.error(f"[VectorMemory] Embedding error: {e}. Falling back to hashing.")

# Simple fallback: map characters to floats

return [float(ord(c)) / 255.0 for c in text][:128]

def add(self, item: Any) -> None:

"""Add an item to both short-term and long-term memory with embedding."""

self.short\_term\_memory.append(item)

key = str(len(self.long\_term\_memory) + 1)

embedding = self.\_embed(str(item))

self.long\_term\_memory[key] = {'item': item, 'embedding': embedding}

logging.debug(f"[VectorMemory] Added item {key}: {item}")

def get\_recent(self, limit: int = 5) -> List[Any]:

"""Return the most recent items from memory."""

return list(self.short\_term\_memory)[-limit:]

def search(self, query: str, top\_k: int = 3) -> List[Any]:

"""Return up to `top\_k` items whose embeddings are closest to the query."""

if not self.long\_term\_memory:

return []

query\_vec = self.\_embed(query)

scored: List[tuple[float, Dict[str, Any]]] = []

for info in self.long\_term\_memory.values():

item\_vec = info['embedding']

if np is not None and len(query\_vec) == len(item\_vec):

try:

sim = float(np.dot(query\_vec, item\_vec) / (np.linalg.norm(query\_vec) \* np.linalg.norm(item\_vec) + 1e-8))

except Exception as e:

logging.error(f"[VectorMemory] Similarity error: {e}")

sim = 0.0

else:

sim = len(set(str(info['item'])) & set(query)) / max(len(set(query)), 1)

scored.append((sim, info))

scored.sort(key=lambda x: x[0], reverse=True)

return [info['item'] for sim, info in scored[:top\_k]]

**ultron\_addons/brain\_extension.py**

python

CopyEdit

"""

brain\_extension.py

===================

This module defines advanced brain implementations that build upon the core

`UltronBrain`. The goal of an advanced brain is to enhance the agent's

reasoning capabilities by incorporating richer context from memory,

performing tool suggestion ranking, and supporting asynchronous

communications with external services.

"""

from \_\_future\_\_ import annotations

import logging

from typing import Any, List, Optional

try:

from brain import UltronBrain # type: ignore

except Exception: # pragma: no cover

UltronBrain = None # type: ignore

class AdvancedUltronBrain:

"""An enhanced brain with improved prompt construction and tool ranking."""

def \_\_init\_\_(self, base\_brain: Optional[UltronBrain], memory: Any, tools: List[Any]) -> None:

self.base\_brain = base\_brain

self.memory = memory

self.tools = tools

logging.info("[AdvancedUltronBrain] Initialized with base brain %s", type(base\_brain).\_\_name\_\_ if base\_brain else None)

def \_rank\_tools(self, user\_input: str) -> List[Any]:

"""Rank available tools based on how well they match the user input."""

scores = []

lower\_input = user\_input.lower()

for tool in self.tools:

score = 0

if hasattr(tool, 'name') and tool.name.lower() in lower\_input:

score += 1

if hasattr(tool, 'description'):

words = set(tool.description.lower().split())

score += len(words & set(lower\_input.split())) / max(len(words), 1)

scores.append((score, tool))

scores.sort(key=lambda x: x[0], reverse=True)

return [tool for \_, tool in scores]

async def plan\_and\_act(self, user\_input: str, progress\_callback: Optional[Any] = None) -> str:

"""Plan and act on a user input using enhanced context."""

context = []

try:

if hasattr(self.memory, 'search'):

context = self.memory.search(user\_input, top\_k=3)

elif hasattr(self.memory, 'get\_recent'):

context = self.memory.get\_recent(3)

except Exception as e:

logging.error(f"[AdvancedUltronBrain] Error retrieving context: {e}")

ranked\_tools = self.\_rank\_tools(user\_input)

if self.base\_brain:

prompt = f"User: {user\_input}\nContext: {context}\n"

prompt += "Available tools (ranked):\n"

for tool in ranked\_tools:

prompt += f"- {tool.name}: {tool.description}\n"

logging.debug(f"[AdvancedUltronBrain] Delegating to base brain with prompt:\n{prompt}")

return await self.base\_brain.query\_llm(prompt, progress\_callback=progress\_callback)

logging.info("[AdvancedUltronBrain] No base brain available, responding with memory context")

if context:

return f"I recall {context[0]} related to your query."

return "I'm sorry, I don't have enough context to assist with that."

**ultron\_addons/tools.py**

python

CopyEdit

"""

tools.py

========

This module defines additional tools that can be plugged into the Ultron

Agent. Tools follow a simple protocol defined by the base `Tool` class

in the core repository: they expose a `name`, `description`, optional

`parameters`, and implement two methods: `match` and `execute`.

"""

from \_\_future\_\_ import annotations

import json

import logging

import requests

from typing import Any

try:

from tools.base import Tool # type: ignore

except Exception:

# Minimal fallback base class

class Tool: # type: ignore

name = "Tool"

description = "Base fallback tool"

parameters: dict[str, Any] = {}

def match(self, command: str) -> bool:

return False

def execute(self, command: str) -> str:

return ""

class SearchTool(Tool):

"""A simple web search tool using DuckDuckGo's instant answer API."""

name = "search\_web"

description = "Search the web for a query and return a brief summary."

parameters = {

"type": "object",

"properties": {

"query": {

"type": "string",

"description": "The search query to look up"

}

},

"required": ["query"],

}

def match(self, command: str) -> bool:

return command.lower().startswith("search")

def execute(self, query: str, \*\*kwargs: Any) -> str:

logging.debug(f"[SearchTool] Executing search for query: {query}")

try:

params = {

"q": query,

"format": "json",

"no\_html": 1,

"skip\_disambig": 1,

}

response = requests.get("https://api.duckduckgo.com/", params=params, timeout=10)

data = response.json()

summary = data.get("Abstract") or next(

(topic.get("Text") for topic in data.get("RelatedTopics", []) if topic.get("Text")),

"No summary available."

)

return summary[:500]

except Exception as e:

logging.error(f"[SearchTool] Error during search: {e}")

return "An error occurred while searching."

class CalculatorTool(Tool):

"""A simple calculator tool that evaluates arithmetic expressions."""

name = "calculate"

description = "Evaluate a basic arithmetic expression and return the result."

parameters = {

"type": "object",

"properties": {

"expression": {

"type": "string",

"description": "A mathematical expression to evaluate"

}

},

"required": ["expression"],

}

def match(self, command: str) -> bool:

return command.lower().startswith("calculate")

def execute(self, expression: str, \*\*kwargs: Any) -> str:

logging.debug(f"[CalculatorTool] Evaluating expression: {expression}")

# Define a restricted set of builtins

allowed\_names = {k: v for k, v in vars(\_\_builtins\_\_).items() if k in ("abs", "round")}

try:

result = eval(expression, {"\_\_builtins\_\_": allowed\_names}, {})

return str(result)

except Exception as e:

logging.error(f"[CalculatorTool] Error evaluating expression: {e}")

return "Invalid expression."

**ultron\_addons/evolution.py**

python

CopyEdit

"""

evolution.py

============

Utilities for managing the evolutionary growth of the Ultron Agent. An

`EvolutionManager` tracks registered connectors, tools and brain

extensions, and can dynamically update the agent configuration.

"""

from \_\_future\_\_ import annotations

import logging

from typing import Any, Dict, List, Optional

class EvolutionManager:

"""Manager for coordinating connectors, tools and brain evolution."""

def \_\_init\_\_(self) -> None:

self.connectors: List[Any] = []

self.tools: List[Any] = []

self.brain\_extension: Optional[Any] = None

def register\_connector(self, connector: Any) -> None:

logging.debug(f"[EvolutionManager] Registering connector: {connector}")

self.connectors.append(connector)

def register\_tool(self, tool: Any) -> None:

logging.debug(f"[EvolutionManager] Registering tool: {tool}")

self.tools.append(tool)

def set\_brain\_extension(self, brain\_extension: Any) -> None:

logging.debug(f"[EvolutionManager] Setting brain extension: {brain\_extension}")

self.brain\_extension = brain\_extension

def summary(self) -> Dict[str, Any]:

"""Return a summary of the current evolutionary state."""

return {

"connectors": [type(c).\_\_name\_\_ for c in self.connectors],

"tools": [type(t).\_\_name\_\_ for t in self.tools],

"brain\_extension": type(self.brain\_extension).\_\_name\_\_ if self.brain\_extension else None,

}

def evolve(self) -> None:

"""Perform a simple evolution step.

In a real system this might fetch updates from a server, load new

plugins from disk, or retrain a model.

"""

logging.info("[EvolutionManager] Starting evolution cycle")

# Example: reconnect all connectors

for conn in self.connectors:

try:

conn.connect()

except Exception as e:

logging.error(f"[EvolutionManager] Failed to connect using {conn}: {e}")

logging.info("[EvolutionManager] Evolution cycle complete")

These files replicate exactly what was in the add‑on package I attempted to send earlier. They provide new connectors, memory, brain, tools and an evolution manager. Once you’ve created the ultron\_addons folder and added these files, you can integrate them into your Ultron agent by importing and registering the relevant classes with your configuration.

Bottom of Form