import os

import json

import sys

import requests

import asyncio

import time

import logging

# Configure debug logging for relevant libraries

logging.getLogger("telegram.ext").setLevel(logging.DEBUG)

logging.getLogger("httpx").setLevel(logging.DEBUG)

from datetime import datetime, timedelta

from decimal import Decimal, ROUND\_HALF\_UP

import pytz

import logging

from openai import OpenAI as Client

from telegram.ext import (

Application,

CommandHandler,

ContextTypes,

MessageHandler,

filters,

ChatJoinRequestHandler

)

from telegram import Update, InlineKeyboardButton, InlineKeyboardMarkup

import openai

from flask import Flask, jsonify

import concurrent.futures

from concurrent.futures import ThreadPoolExecutor, as\_completed

# --- Logging Functions ---

def log\_message(message, component="BOT"):

timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

print(f"[{timestamp}] [{component}] {message}")

def log\_error(message, component="BOT"):

timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

print(f"[{timestamp}] [{component}] ERROR: {message}")

# Append error to a log file

try:

with open("error\_log.txt", "a") as f:

f.write(f"{datetime.now().isoformat()}: {message}\n")

except Exception as e:

print(f"[{timestamp}] [LOGGING] CRITICAL ERROR: Could not write to error\_log.txt: {str(e)}")

# --- Initialize DeepSeek client ---

try:

if os.getenv('DEEPSEEK\_API\_KEY'):

deepseek\_client = Client(

api\_key=os.getenv('DEEPSEEK\_API\_KEY'),

base\_url="https://api.deepseek.com"

)

log\_message("DeepSeek API client initialized", component="INIT")

else:

log\_message("DeepSeek API key not found in environment", component="INIT")

deepseek\_client = None

except Exception as e:

log\_error(f"Failed to initialize DeepSeek API client: {str(e)}", component="INIT")

deepseek\_client = None

# --- Flask App Initialization ---

app = Flask(\_\_name\_\_)

flask\_app = app # Ensure the Flask app is accessible at module level for Gunicorn

# --- Telegram & OpenAI Imports already above ---

# (no duplicate logging functions now)

# --- Command Handler Wrapper ---

def command\_handler\_wrapper(func):

"""Wrapper for command handlers with error handling"""

async def wrapper(update, context):

try:

await func(update, context)

except Exception as e:

log\_error(f"Command {func.\_\_name\_\_} failed: {str(e)}", component="HANDLER")

await update.message.reply\_text("Sorry, command execution failed. Please try again.")

return wrapper

# --- Global Variables & Configurations ---

START\_TIME = datetime.now()

SUBSCRIBERS\_FILE = "subscribers.json"

BLOCKED\_USERS\_FILE = "blocked\_users.json"

try:

with open(BLOCKED\_USERS\_FILE, 'r') as f:

BLOCKED\_USERS = set(json.load(f))

except FileNotFoundError:

log\_message("blocked\_users.json not found, creating a new one.", component="INIT")

BLOCKED\_USERS = set()

with open(BLOCKED\_USERS\_FILE, 'w') as f:

json.dump(list(BLOCKED\_USERS), f)

except json.JSONDecodeError:

log\_error("Error decoding blocked\_users.json. Initializing with an empty set.", component="INIT")

BLOCKED\_USERS = set()

USER\_DATABASE\_FILE = "user\_database.json"

USER\_QUESTIONS\_FILE = "user\_questions.json"

# Validate required environment variables

TELEGRAM\_BOT\_TOKEN = os.environ.get("TELEGRAM\_BOT\_TOKEN")

CHANNEL\_ID = os.environ.get("CHANNEL\_ID")

if not TELEGRAM\_BOT\_TOKEN:

log\_error("CRITICAL: TELEGRAM\_BOT\_TOKEN is not set in environment variables", component="INIT")

sys.exit(1)

if not CHANNEL\_ID:

log\_message("WARNING: CHANNEL\_ID is not set in environment variables", component="INIT")

log\_message(f"Bot token validation: {'Present' if TELEGRAM\_BOT\_TOKEN else 'Missing'}", component="INIT")

log\_message(f"Channel ID validation: {'Present' if CHANNEL\_ID else 'Missing'}", component="INIT")

# --- Production Adjustments ---

NODE\_SERVICE\_URL = "http://0.0.0.0:3000"

log\_message(f"Internal Node.js service URL set to: {NODE\_SERVICE\_URL}", component="INIT")

PYTHON\_PORT = int(os.environ.get("PORT", 5000))

PYTHON\_HOST = "0.0.0.0" # Bind to all interfaces

log\_message(f"Flask will run on {PYTHON\_HOST}:{PYTHON\_PORT}", component="INIT")

REPL\_SLUG = os.getenv('REPL\_SLUG')

REPL\_OWNER = os.getenv('REPL\_OWNER')

KEEP\_ALIVE\_URL = f"{NODE\_SERVICE\_URL}/health" # Adjusted to use NODE\_SERVICE\_URL and assuming a /health endpoint

log\_message(f"Keep-alive check URL set to: {KEEP\_ALIVE\_URL}", component="INIT")

FOOTER = "\n\n[Swap USDT to SOL now](http://bit.ly/4kfCgW7)" # Placeholder link

# --- OpenAI & DeepSeek Configurations ---

openai.api\_key = os.getenv('OPENAI\_API\_KEY') # Standard OpenAI if used

DEEPSEEK\_API\_KEY = os.getenv('DEEPSEEK\_API\_KEY')

deepseek\_client = None

try:

if DEEPSEEK\_API\_KEY:

deepseek\_client = openai.OpenAI(

api\_key=DEEPSEEK\_API\_KEY,

base\_url="https://api.deepseek.com"

)

log\_message("DeepSeek API client initialized", component="INIT")

else:

log\_message("DeepSeek API key (DEEPSEEK\_API\_KEY) not found. /ask command might be limited.", component="INIT")

except Exception as e:

log\_error(f"Failed to initialize DeepSeek API client: {str(e)}", component="INIT")

user\_context = {}

executor = ThreadPoolExecutor(max\_workers=10) # Adjust workers as needed

# --- Helper Functions ---

def append\_footer(message: str) -> str:

"""Appends the standard footer to a message."""

return message + FOOTER

def append\_ask\_note(message: str) -> str:

"""Appends a note prompting users to use /ask for product questions."""

note = "\n\n\*For product-related questions, please use the /ask command.\*"

return message + note

def format\_price(price):

"""Formats a price value to one decimal place with a dollar sign."""

try:

price\_decimal = Decimal(str(price))

formatted = price\_decimal.quantize(Decimal("0.1"), rounding=ROUND\_HALF\_UP)

return f"${formatted}"

except Exception as e:

log\_error(f"Error formatting price '{price}': {str(e)}", component="HELPER")

return f"${str(price)}"

def format\_token\_price(raw\_price, token\_symbol="Token"):

"""Formats a token price to one decimal place."""

try:

p = float(raw\_price)

return f"${p:.1f}"

except (ValueError, TypeError) as e:

log\_error(f"Error formatting token price '{raw\_price}' for {token\_symbol}: {str(e)}", component="HELPER")

return f"${raw\_price}"

# --- Database Utilities ---

logging.basicConfig(level=logging.INFO)

db\_logger = logging.getLogger('database')

try:

from db\_utils import (

load\_subscribers, save\_subscriber, remove\_subscriber,

load\_user\_database, update\_user\_activity,

save\_user\_question, load\_user\_questions,

get\_db\_stats, backup\_database, export\_questions\_to\_txt,

track\_user\_question, init\_db

)

# Initialize database connection and tables

if init\_db():

log\_message("Database initialized successfully", component="INIT")

else:

log\_error("Database initialization failed, using file fallback", component="INIT")

except ImportError as e:

log\_error(f"Failed to import from db\_utils.py: {str(e)}", component="INIT")

def load\_subscribers(): return set()

def save\_subscriber(chat\_id): pass

def remove\_subscriber(chat\_id): pass

def load\_user\_database(): return {}

def update\_user\_activity(chat\_id, user\_data, command): pass

def save\_user\_question(chat\_id, question, answer): pass

def load\_user\_questions(): return []

def get\_db\_stats(): return {}

def backup\_database(): pass

def export\_questions\_to\_txt(): pass

def track\_user\_question(chat\_id, question\_type): pass

def track\_user\_activity(update, command=None):

"""Tracks user activity and updates the database."""

if not update or not update.effective\_user or not update.effective\_chat:

log\_message("Attempted to track activity with invalid update object.", component="TRACKING")

return None

chat\_id = update.effective\_chat.id

user = update.effective\_user

user\_data = {

'chat\_id': chat\_id,

'username': user.username or "N/A",

'first\_name': user.first\_name or "N/A",

'last\_name': user.last\_name or "",

'language\_code': user.language\_code or "N/A",

'is\_subscribed': chat\_id in subscribers

}

update\_user\_activity(chat\_id, user\_data, command)

return user\_data

subscribers = load\_subscribers() or set()

log\_message(f"Loaded {len(subscribers)} subscribers from file.", component="INIT")

# --- Node.js Service Interaction ---

\_pools\_cache = {

'data': None,

'timestamp': 0,

'duration': 30 # Cache duration in seconds

}

\_http\_session = requests.Session()

adapter = requests.adapters.HTTPAdapter(

pool\_connections=20,

pool\_maxsize=30,

max\_retries=3,

pool\_block=False

)

\_http\_session.mount('http://', adapter)

\_http\_session.mount('https://', adapter)

\_http\_session.timeout = 30 # Increase timeout to 30 seconds

def fetch\_pools\_data():

"""Fetches pool data from the internal Node.js service, using cache."""

now = time.time()

if \_pools\_cache['data'] and (now - \_pools\_cache['timestamp'] < \_pools\_cache['duration']):

log\_message("Returning cached pools data.", component="NODE\_API")

return \_pools\_cache['data']

url = f"{NODE\_SERVICE\_URL}/pools"

max\_retries = 3

retry\_delay = 1

fallback\_data = {

"topAPR": [{"id": "fallback\_id", "tokenPair": "SOL/USDC", "day": {"apr": 10.0}, "week": {"apr": 11.0}, "month": {"apr": 12.0}, "tvl": 1000000, "price": 100.0, "priceToken": "SOL"}],

"mandatory": [{"id": "fallback\_mand", "tokenPair": "SOL/USDT", "day": {"apr": 9.0}, "week": {"apr": 10.0}, "month": {"apr": 11.0}, "tvl": 900000, "price": 100.0, "priceToken": "SOL"}]

}

for attempt in range(max\_retries):

try:

log\_message(f"Fetching live pools data from {url} (attempt {attempt+1}/{max\_retries})", component="NODE\_API")

response = \_http\_session.get(url, timeout=16)

if response.status\_code == 200:

data = response.json()

if isinstance(data, dict) and "topAPR" in data and "mandatory" in data:

\_pools\_cache['data'] = data

\_pools\_cache['timestamp'] = now

log\_message("Successfully fetched and cached live pools data.", component="NODE\_API")

return data

else:

log\_error(f"API at {url} returned invalid data structure: {str(data)[:200]}", component="NODE\_API")

if attempt == max\_retries - 1:

log\_error("Using fallback data due to invalid structure.", component="NODE\_API")

return fallback\_data

continue

else:

log\_error(f"API at {url} returned non-200 status: {response.status\_code}", component="NODE\_API")

response.raise\_for\_status()

except requests.exceptions.Timeout:

log\_error(f"API connection to {url} timed out on attempt {attempt+1}.", component="NODE\_API")

except requests.exceptions.ConnectionError:

log\_error(f"API connection refused at {url} on attempt {attempt+1}. Is Node.js running?", component="NODE\_API")

except requests.exceptions.RequestException as e:

log\_error(f"API request error to {url} on attempt {attempt+1}: {str(e)}", component="NODE\_API")

except json.JSONDecodeError:

log\_error(f"Failed to decode JSON response from {url}", component="NODE\_API")

except Exception as e:

log\_error(f"Unexpected error fetching pools data on attempt {attempt+1}: {str(e)}", component="NODE\_API")

if attempt < max\_retries - 1:

log\_message(f"Retrying pools data fetch in {retry\_delay} seconds...", component="NODE\_API")

time.sleep(retry\_delay)

retry\_delay \*= 2

else:

log\_error(f"Failed to fetch live pools data from {url} after {max\_retries} attempts. Using fallback data.", component="NODE\_API")

return fallback\_data

return fallback\_data

# --- Dynamic Admin Check Helper ---

async def is\_group\_admin(update: Update, context: ContextTypes.DEFAULT\_TYPE) -> bool:

"""Checks if the user sending a command in a group is an admin."""

if not update.effective\_chat or not update.effective\_user:

return False

chat\_id = update.effective\_chat.id

user\_id = update.effective\_user.id

if update.effective\_chat.type not in ["group", "supergroup"]:

return False

try:

admins = await context.bot.get\_chat\_administrators(chat\_id)

admin\_ids = {admin.user.id for admin in admins}

is\_admin = user\_id in admin\_ids

log\_message(f"Admin check for user {user\_id} in chat {chat\_id}: {'Is admin' if is\_admin else 'Not admin'}", component="AUTH")

return is\_admin

except Exception as e:

log\_error(f"Failed to get chat administrators for chat {chat\_id}: {str(e)}", component="AUTH")

return False

# --- New Handler for Chat Join Requests ---

async def join\_request\_handler(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles new users requesting to join the chat."""

if not update.chat\_join\_request:

return

request = update.chat\_join\_request

user = request.from\_user

chat = request.chat

chat\_id = chat.id

user\_id = user.id

log\_message(f"Received join request from user {user\_id} ({user.username or user.first\_name}) for chat {chat\_id} ({chat.title or 'N/A'})", component="JOIN")

if user.is\_bot:

log\_message(f"Declining join request from bot {user\_id}.", component="JOIN")

try:

await context.bot.decline\_chat\_join\_request(chat\_id=chat\_id, user\_id=user\_id)

except Exception as e:

log\_error(f"Failed to decline bot join request for {user\_id}: {str(e)}", component="JOIN")

return

is\_suspicious = (not user.has\_profile\_photo) or (user.username and "bot" in user.username.lower())

if is\_suspicious:

log\_message(f"Suspicious join request from user {user\_id}. Flagging for manual review.", component="JOIN")

owner\_chat\_id = os.getenv("OWNER\_CHAT\_ID")

if owner\_chat\_id:

try:

message = (

f"Â¡âš ï¸ \*Suspicious Join Request\* âš ï¸\n\n"

f"Chat: {chat.title or chat\_id}\n"

f"User: {user.first\_name} {user.last\_name or ''} (@{user.username or 'N/A'})\n"

f"User ID: `{user\_id}`\n"

f"Reason: {'No profile photo' if not user.has\_profile\_photo else 'Username contains bot'}\n\n"

f"Please review and manually approve/decline."

)

await context.bot.send\_message(chat\_id=owner\_chat\_id, text=message, parse\_mode="Markdown")

except Exception as e:

log\_error(f"Failed to send suspicious join notification to owner {owner\_chat\_id}: {str(e)}", component="JOIN")

return

try:

log\_message(f"Approving join request for user {user\_id}.", component="JOIN")

await context.bot.approve\_chat\_join\_request(chat\_id=chat\_id, user\_id=user\_id)

except Exception as e:

log\_error(f"Failed to approve join request for user {user\_id} in chat {chat\_id}: {str(e)}", component="JOIN")

# --- Telegram Command Handlers ---

async def start(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /start command."""

log\_message("Entered /start handler", component="HANDLER\_ENTRY")

chat\_id = update.effective\_chat.id

user = update.effective\_user

username = user.username or user.first\_name

log\_message(f"Command /start executed by {username} (ID: {user.id}, Chat: {chat\_id})", component="TELEGRAM")

track\_user\_activity(update, "start")

welcome\_message = (

"ðŸ’¡ \*Welcome to the FiLot AI-powered Investment Assistant!\*\n\n"

"ðŸš€ I help you track real-time crypto earnings and updates â€“ think of me as your AI assistant for maximizing returns through FiLot's liquidity pool optimization.\n\n"

"ðŸ’¡ \*Why care?\*\n"

"ðŸ¦ Banks offer low interest (0.5-2%/year). FiLot's AI finds liquidity pools potentially earning much more (10-200%+) based on market conditions, automatically and safely.\n\n"

"ðŸ“ \*How does it work?\*\n"

"ðŸ” FiLot AI scans the market, predicts top pools, and helps you invest easily. No complex DeFi knowledge needed!\n\n"

"â± \*See potential earnings?\*\n"

"ðŸ’° Use /simulate to estimate returns from providing liquidity. Just enter an amount!\n\n"

"â“ \*What's a liquidity pool?\*\n"

"Itâ€™s a shared pot of crypto. When trades use the pool, contributors earn fees. More contribution = more earnings!\n\n"

"âœ¨ \*FiLot Makes It Easy & Safe\*\n"

"â€¢ AI-Powered Yield Optimization\n"

"â€¢ Risk Protection (focus on stable pools)\n"

"â€¢ One-Click Investing (coming soon)\n\n"

"ðŸ“¢ \*Stay Updated\*\n"

"Type /subscribe for automatic updates, /info for today's top earnings, or /help for commands.\n\n"

"ðŸ¤– \*Have questions?\* Just type them, or use /ask for specific product info.\n\n"

"ðŸ’¡ FiLot is launching soon! Get ready to let AI grow your money smarter.\n\n"

"\*Token Contract Address (CA)\*:\n`Bpz8btet2EVDzjjHZAaqwjTzE5p62H9Sb5RuKVNBpump`\n\n"

"\*Buy LA! Token:\*\nhttps://pump.fun/coin/Bpz8btet2EVDzjjHZAaqwjTzE5p62H9Sb5RuKVNBpump"

)

final\_message = append\_footer(welcome\_message)

try:

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

except Exception as e:

log\_error(f"Failed to send /start message to chat {chat\_id}: {str(e)}", component="TELEGRAM")

async def subscribe(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /subscribe command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /subscribe executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "subscribe")

if chat\_id not in subscribers:

subscribers.add(chat\_id)

save\_subscriber(chat\_id)

message = (

"âœ… You're now subscribed to updates! I'll send you the latest earnings and price updates for selected crypto investments. "

"To stop, type /unsubscribe."

)

log\_message(f"Chat {chat\_id} successfully subscribed.", component="TELEGRAM")

else:

message = "âœ… You are already subscribed!"

log\_message(f"Chat {chat\_id} tried to subscribe again.", component="TELEGRAM")

final\_message = append\_footer(message)

try:

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

except Exception as e:

log\_error(f"Failed to send /subscribe confirmation to chat {chat\_id}: {str(e)}", component="TELEGRAM")

async def unsubscribe(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /unsubscribe command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /unsubscribe executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "unsubscribe")

if chat\_id in subscribers:

subscribers.remove(chat\_id)

remove\_subscriber(chat\_id)

message = (

"ðŸš« You've unsubscribed from updates. You can still check info with /info. "

"To re-subscribe, type /subscribe."

)

log\_message(f"Chat {chat\_id} successfully unsubscribed.", component="TELEGRAM")

else:

message = "â„¹ï¸ You are not currently subscribed."

log\_message(f"Chat {chat\_id} tried to unsubscribe but wasn't subscribed.", component="TELEGRAM")

final\_message = append\_footer(message)

try:

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

except Exception as e:

log\_error(f"Failed to send /unsubscribe confirmation to chat {chat\_id}: {str(e)}", component="TELEGRAM")

async def info(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /info command, fetching and displaying pool data."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /info executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "info")

await update.message.reply\_chat\_action(action="typing")

try:

data = fetch\_pools\_data()

top\_apr = data.get("topAPR", [])

mandatory = data.get("mandatory", [])

message = "ðŸ“ˆ Latest Crypto Investment Update:\n\n"

if not top\_apr and not mandatory:

message += "â³ Currently unable to fetch live pool data. Please try again later."

else:

if top\_apr:

message += "Best Performing Investments Today:\n"

for pool in top\_apr[:3]:

pool\_id = pool.get('id', 'N/A')

token\_pair = pool.get("tokenPair", "Unknown Pair")

day\_apr = float(pool.get('day', {}).get('apr', 0))

week\_apr = float(pool.get('week', {}).get('apr', 0))

month\_apr = float(pool.get('month', {}).get('apr', 0))

tvl = pool.get('tvl', 0)

price = pool.get("price", 0)

price\_token = pool.get("priceToken", "")

current\_price\_str = (f"${price:.1f} per {price\_token}"

if price\_token

else f"${price:.1f} per Token")

message += (

f"â€¢ Pool ID: ðŸ“‹ `{pool\_id}`\n"

f" Token Pair: {token\_pair}\n"

f" 24h APR: {day\_apr:.2f}%\n"

f" 7d APR: {week\_apr:.2f}%\n"

f" 30d APR: {month\_apr:.2f}%\n"

f" TVL (USD): ${tvl:,.2f}\n"

f" Current Price (USD): {current\_price\_str}\n\n"

)

if mandatory:

message += "Top Stable Investments (e.g., SOL-USDC / SOL-USDT):\n"

for pool in mandatory:

pool\_id = pool.get('id', 'N/A')

token\_pair = pool.get("tokenPair", "Unknown Pair")

day\_apr = float(pool.get('day', {}).get('apr', 0))

week\_apr = float(pool.get('week', {}).get('apr', 0))

month\_apr = float(pool.get('month', {}).get('apr', 0))

tvl = pool.get('tvl', 0)

price = pool.get("price", 0)

price\_token = pool.get("priceToken", "")

current\_price\_str = (f"${price:.1f} per {price\_token}"

if price\_token

else f"${price:.1f} per Token")

message += (

f"â€¢ Pool ID: ðŸ“‹ `{pool\_id}`\n"

f" Token Pair: {token\_pair}\n"

f" 24h APR: {day\_apr:.2f}%\n"

f" 7d APR: {week\_apr:.2f}%\n"

f" 30d APR: {month\_apr:.2f}%\n"

f" TVL (USD): ${tvl:,.2f}\n"

f" Current Price (USD): {current\_price\_str}\n\n"

)

message += "\nWant to see your potential earnings? Try /simulate amount (default is $1000)."

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown", disable\_web\_page\_preview=True)

log\_message(f"/info data sent successfully to chat {chat\_id}.", component="TELEGRAM")

except Exception as e:

log\_error(f"/info command failed for chat {chat\_id}: {str(e)}", component="TELEGRAM")

err\_message = append\_footer("â³ Sorry, I couldn't fetch the latest investment info right now. Please try again later.")

await update.message.reply\_text(err\_message, parse\_mode="Markdown")

async def simulate(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /simulate command to estimate earnings."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /simulate executed by Chat: {chat\_id} with args: {context.args}", component="TELEGRAM")

track\_user\_activity(update, "simulate")

principal = 1000.0

if context.args:

try:

amount\_str = context.args[0].replace('$', '').replace(',', '')

principal = float(amount\_str)

if principal <= 0:

await update.message.reply\_text("â³ Please enter a positive amount to simulate (e.g., /simulate 500).")

return

except ValueError:

await update.message.reply\_text("â³ Invalid amount. Please use a number (e.g., /simulate 500).")

return

await update.message.reply\_chat\_action(action="typing")

try:

data = fetch\_pools\_data()

all\_pools = {pool['id']: pool for pool in data.get("topAPR", []) + data.get("mandatory", []) if 'id' in pool}.values()

if not all\_pools:

await update.message.reply\_text(append\_footer("â³ Could not fetch pool data for simulation. Please try /info or try again later."))

return

message = f"ðŸš€ Simulation for an Investment of ${principal:,.2f}:\n\n"

pools\_to\_simulate = list(all\_pools)[:5]

for pool in pools\_to\_simulate:

pool\_id = pool.get('id', 'N/A')

token\_pair = pool.get('tokenPair', 'Unknown')

month\_apr = float(pool.get('month', {}).get('apr', 0))

week\_apr = float(pool.get('week', {}).get('apr', 0))

day\_apr = float(pool.get('day', {}).get('apr', 0))

base\_apr = month\_apr if month\_apr > 0 else (week\_apr if week\_apr > 0 else day\_apr)

if base\_apr <= 0:

continue

daily\_earnings = principal \* (base\_apr / 100) / 365

weekly\_earnings = daily\_earnings \* 7

monthly\_earnings = daily\_earnings \* 30

annual\_earnings = daily\_earnings \* 365

message += (

f"â€¢ Pool ID: ðŸ“‹ `{pool\_id}`\n"

f" Token Pair: {token\_pair}\n"

f" - Daily Earnings: ${daily\_earnings:,.2f}\n"

f" - Weekly Earnings: ${weekly\_earnings:,.2f}\n"

f" - Monthly Earnings: ${monthly\_earnings:,.2f}\n"

f" - Annual Earnings: ${annual\_earnings:,.2f}\n\n"

)

message += "\nDisclaimer: The numbers above are estimations and actual earnings may vary."

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown", disable\_web\_page\_preview=True)

log\_message(f"/simulate calculation sent to chat {chat\_id}.", component="TELEGRAM")

except Exception as e:

log\_error(f"/simulate command failed for chat {chat\_id}: {str(e)}", component="TELEGRAM")

err\_message = append\_footer("â³ An error occurred during the simulation. Please try again.")

await update.message.reply\_text(err\_message, parse\_mode="Markdown")

async def status(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /status command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /status executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "status")

uptime = datetime.now() - START\_TIME

hours, remainder = divmod(uptime.total\_seconds(), 3600)

minutes, seconds = divmod(remainder, 60)

uptime\_str = f"{int(hours):02}:{int(minutes):02}:{int(seconds):02}"

users\_data = load\_user\_database()

total\_users = len(users\_data)

node\_status = "â³ Unknown"

try:

response = \_http\_session.get(f"{NODE\_SERVICE\_URL}/health", timeout=3)

if response.status\_code == 200:

node\_status = "âœ… Online"

else:

node\_status = f"â³ Status {response.status\_code}"

except requests.exceptions.RequestException:

node\_status = "â³ Offline"

except Exception:

node\_status = "â³ Error Checking"

message = (

f"ðŸ“Š \*Bot Status\*\n\n"

f"âœ… Bot Service: Online\n"

f"â³ Internal API Service: {node\_status}\n"

f"â± Uptime: {uptime\_str}\n"

f"ðŸš€ Active Subscriptions: {len(subscribers)}\n"

f"ðŸ“Š Total Users Tracked: {total\_users}\n"

)

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

log\_message(f"/status response sent to chat {chat\_id}. Node status: {node\_status}", component="TELEGRAM")

async def joinpool\_command(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /joinpool command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /joinpool executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "joinpool")

message = (

"ðŸ”„ \*Pool Joining Feature -Coming Soon!\*\n\n"

"The direct pool joining feature is currently under development and will be available soon!\n\n"

"In the meantime, you can:\n"

"â€¢ Use /info to view available pools and their performance\n"

"â€¢ Try /simulate to calculate potential earnings\n"

"â€¢ Stay updated by using /subscribe\n\n"

"We're working hard to bring you seamless pool integration. Thank you for your patience! ðŸš€"

)

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

async def help\_command(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /help command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /help executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "help")

message = (

"âœ… How can I help? Here's what you can do:\n\n"

"/start - Welcome message and introduction\n"

"/subscribe - Subscribe to updates\n"

"/unsubscribe - Unsubscribe from updates\n"

"/info - Latest crypto investment info\n"

"/simulate [amount] - Simulate potential earnings (default $1000)\n"

"/status - Bot status and subscriber count\n"

"/FAQ - Frequently asked questions\n"

"/Social - Follow our official social channels\n"

"/ask [question] - Ask product-related questions\n"

"/joinpool <pool\_id> <wallet\_address> - Execute a join pool transaction\n\n"

"ðŸ’¬ You can also simply type your questions directly!"

)

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

async def faq(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /faq command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /faq executed byChat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "faq")

message = (

"â“ \*Frequently Asked Questions\*\n\n"

"\*What does this bot do?\*\n"

"It helps track crypto earnings from AI-optimized liquidity pools and provides info on FiLot.\n\n"

"\*How do I earn with liquidity pools?\*\n"

"Contribute crypto to earn trading fees. FiLot aims to find high-yield, safer pools.\n\n"

"\*Is it risky?\*\n"

"All investments have risk. FiLot focuses on stable pools (like SOL-USDC) and uses AI to manage risk, but losses are possible.\n\n"

"\*What's Impermanent Loss (IL)?\*\n"

"Value changes in your deposited tokens compared to just holding them. AI aims to minimize this by selecting suitable pools.\n\n"

"\*What's APR?\*\n"

"Annual Percentage Rate - estimated yearly return. Pool APRs can be high (10-200%+) but fluctuate.\n\n"

"\*How do updates work?\*\n"

"Use /subscribe for automatic news. Use /unsubscribe to stop.\n\n"

"\*How does /simulate work?\*\n"

"It estimates earnings based on recent APRs: `Earnings = Investment \* (APR/100) \* Time`.\n\n"

"\*When is FiLot launching?\*\n"

"Coming soon! Use /subscribe for announcements.\n\n"

"\*How do I ask specific questions?\*\n"

"Use `/ask Your question here...` for product details, or just type general questions.\n\n"

)

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown", disable\_web\_page\_preview=True)

async def social(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /social command."""

chat\_id = update.effective\_chat.id

log\_message(f"Command /social executed by Chat: {chat\_id}", component="TELEGRAM")

track\_user\_activity(update, "social")

message = (

"ðŸ¤ \*Connect With Us!\*\n\n"

"Stay updated and join the community on our official channels:\n\n"

"ðŸ¦ \*\*X (Twitter):\*\* [@CrazyRichLA](https://x.com/crazyrichla)\n"

"ðŸ“¸ \*\*Instagram:\*\* [@CrazyRichLA](https://www.instagram.com/crazyrichla)\n"

"ðŸŒ \*\*Website:\*\* [CrazyRichLA](https://crazyrichla.replit.app)\n\n"

"Follow us for the latest news and launch announcements!"

)

final\_message = append\_footer(message)

await update.message.reply\_text(final\_message, parse\_mode="Markdown", disable\_web\_page\_preview=True)

# --- Admin Commands ---

async def ban\_command(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""(Admin) Bans a user."""

chat\_id = update.effective\_chat.id

user\_id = update.effective\_user.id

log\_message(f"Command /ban attempt by User: {user\_id} in Chat: {chat\_id}", component="ADMIN")

track\_user\_activity(update, "ban\_attempt")

if not await is\_group\_admin(update, context):

await update.message.reply\_text("ðŸš« Sorry, this command is only for chat administrators.")

return

target\_user\_id = None

if update.message.reply\_to\_message:

target\_user\_id = update.message.reply\_to\_message.from\_user.id

log\_message(f"Admin {user\_id} targeting user {target\_user\_id} via reply for banning.", component="ADMIN")

elif context.args:

try:

target\_user\_id = int(context.args[0])

log\_message(f"Admin {user\_id} targeting user {target\_user\_id} via argument for banning.", component="ADMIN")

except (ValueError, IndexError):

await update.message.reply\_text("â³ Usage: Reply to a user's message or use `/ban <user\_id>`.")

return

else:

await update.message.reply\_text("â³ Usage: Reply to the user you want to ban or use `/ban <user\_id>`.")

return

if target\_user\_id:

if target\_user\_id == context.bot.id:

await update.message.reply\_text("ðŸš« Nice try! I can't ban myself.")

return

if target\_user\_id == user\_id:

await update.message.reply\_text("ðŸš« You can't ban yourself.")

return

try:

target\_member = await context.bot.get\_chat\_member(chat\_id, target\_user\_id)

if target\_member.status in ['administrator', 'creator']:

await update.message.reply\_text("ðŸš« Cannot ban another administrator/creator.")

log\_message(f"Admin {user\_id} failed to ban admin/creator {target\_user\_id}.", component="ADMIN")

return

except Exception as e:

log\_error(f"Could not check target user {target\_user\_id} status before ban: {str(e)}", component="ADMIN")

try:

await context.bot.ban\_chat\_member(chat\_id=chat\_id, user\_id=target\_user\_id)

BLOCKED\_USERS.add(target\_user\_id)

with open(BLOCKED\_USERS\_FILE, 'w') as f:

json.dump(list(BLOCKED\_USERS), f)

await update.message.reply\_text(f"âœ… User {target\_user\_id} has been banned.")

log\_message(f"Admin {user\_id} successfully banned user {target\_user\_id} in chat {chat\_id}.", component="ADMIN")

except Exception as e:

log\_error(f"Failed to ban user {target\_user\_id} in chat {chat\_id}: {str(e)}", component="ADMIN")

await update.message.reply\_text(f"â³ Failed to ban user {target\_user\_id}. Error: {str(e)}")

async def unban\_command(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""(Admin) Unbans a user."""

chat\_id = update.effective\_chat.id

user\_id = update.effective\_user.id

log\_message(f"Command /unban attempt by User: {user\_id} in Chat: {chat\_id}", component="ADMIN")

track\_user\_activity(update, "unban\_attempt")

if not await is\_group\_admin(update, context):

await update.message.reply\_text("ðŸš« Sorry, this command is only for chat administrators.")

return

target\_user\_id = None

if context.args:

try:

target\_user\_id = int(context.args[0])

log\_message(f"Admin {user\_id} targeting user {target\_user\_id} via argument for unbanning.", component="ADMIN")

except (ValueError, IndexError):

await update.message.reply\_text("â³ Usage: `/unban <user\_id>`.")

return

else:

await update.message.reply\_text("â³ Usage: `/unban <user\_id>`.")

return

if target\_user\_id:

try:

await context.bot.unban\_chat\_member(chat\_id=chat\_id, user\_id=target\_user\_id, only\_if\_banned=True)

if target\_user\_id in BLOCKED\_USERS:

BLOCKED\_USERS.remove(target\_user\_id)

with open(BLOCKED\_USERS\_FILE, 'w') as f:

json.dump(list(BLOCKED\_USERS), f)

await update.message.reply\_text(f"âœ… User {target\_user\_id} has been unbanned. They can rejoin.")

log\_message(f"Admin {user\_id} successfully unbanned user {target\_user\_id} in chat {chat\_id}.", component="ADMIN")

except Exception as e:

log\_error(f"Failed to unban user {target\_user\_id} in chat {chat\_id}: {str(e)}", component="ADMIN")

if "user is not found" in str(e).lower() or "user\_not\_participant" in str(e).lower() or "wasn't banned" in str(e).lower():

await update.message.reply\_text(f"ðŸš« User {target\_user\_id} doesn't seem to be banned.")

else:

await update.message.reply\_text(f"â³ Failed to unban user {target\_user\_id}. Error: {str(e)}")

# --- AI /ask Command and Text Handler ---

async def ask\_command(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles the /ask command using DeepSeek or fallback."""

chat\_id = update.effective\_chat.id

user\_question = ' '.join(context.args)

log\_message(f"Command /ask executed by Chat: {chat\_id} with question: '{user\_question}'", component="AI")

track\_user\_activity(update, "ask")

if not user\_question:

await update.message.reply\_text("ðŸš« Please ask a question after the command, like `/ask What is FiLot?`")

return

if not deepseek\_client:

await update.message.reply\_text("â³ The AI assistant (DeepSeek) is currently unavailable. Please try again later.")

log\_error("Attempted /ask command but DeepSeek client is not initialized.", component="AI")

return

from response\_data import get\_predefined\_response

predefined\_answer = get\_predefined\_response(user\_question)

if predefined\_answer and predefined\_answer.strip():

log\_message(f"Using predefined answer for /ask query: {user\_question}", component="PREDEFINED")

final\_message = append\_footer(predefined\_answer)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

return

await update.message.reply\_chat\_action(action="typing")

if chat\_id not in user\_context:

user\_context[chat\_id] = [{"role": "system", "content": "You are FiLot AI, a helpful assistant knowledgeable about FiLot liquidity pools, crypto investments, and the LA! token project. Answer user questions clearly and concisely based on your training data. If you don't know the answer, say so. Keep answers focused on the product and related crypto concepts."}]

user\_context[chat\_id].append({"role": "user", "content": user\_question})

max\_context = 10

if len(user\_context[chat\_id]) > max\_context:

user\_context[chat\_id] = [user\_context[chat\_id][0]] + user\_context[chat\_id][-(max\_context - 1):]

try:

response = await asyncio.get\_event\_loop().run\_in\_executor(

executor,

lambda: deepseek\_client.chat.completions.create(

model="deepseek-chat",

messages=user\_context[chat\_id],

max\_tokens=300,

temperature=0.7

)

)

ai\_response = response.choices[0].message.content.strip()

user\_context[chat\_id].append({"role": "assistant", "content": ai\_response})

final\_message = append\_footer(ai\_response)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

log\_message(f"Successfully processed /ask command for chat {chat\_id}.", component="AI")

save\_user\_question(chat\_id, user\_question, ai\_response)

track\_user\_question(chat\_id, "ask\_command")

except openai.APIConnectionError as e:

log\_error(f"DeepSeek API connection error for /ask: {str(e)}", component="AI")

await update.message.reply\_text("â³ Sorry, I couldn't connect to the AI assistant right now. Please check your connection or try again later.")

except openai.RateLimitError as e:

log\_error(f"DeepSeek API rate limit exceeded for /ask: {str(e)}", component="AI")

await update.message.reply\_text("âš¡ The AI assistant is currently busy. Please try again in a moment.")

except openai.APIStatusError as e:

log\_error(f"DeepSeek API status error for /ask (Code: {e.status\_code}): {e.response}", component="AI")

await update.message.reply\_text(f"â³ An error occurred with the AI assistant (Code: {e.status\_code}). Please try again.")

except Exception as e:

log\_error(f"General error processing /ask command for chat {chat\_id}: {str(e)}", component="AI")

await update.message.reply\_text("â³ An unexpected error occurred while processing your request.")

async def text\_message\_handler(update: Update, context: ContextTypes.DEFAULT\_TYPE):

"""Handles regular text messages, potentially routing to AI or specific logic."""

chat\_id = update.effective\_chat.id

user\_message = update.message.text

log\_message(f"Received text message from Chat: {chat\_id}: '{user\_message[:50]}...'", component="TELEGRAM")

track\_user\_activity(update, "text\_message")

if len(user\_message.split()) < 3 or user\_message.lower() in ["hi", "hello", "thanks", "ok", "gm", "gn"]:

log\_message(f"Ignoring short/greeting message from {chat\_id}.", component="TELEGRAM")

return

from response\_data import get\_predefined\_response

# Normalize message for better matching

normalized\_message = user\_message.lower().strip()

predefined\_answer = get\_predefined\_response(normalized\_message)

if predefined\_answer and predefined\_answer.strip():

log\_message(f"Using predefined answer for text message: {user\_message[:50]}...", component="PREDEFINED")

final\_message = append\_footer(predefined\_answer)

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

track\_user\_question(chat\_id, "predefined\_response")

return

if deepseek\_client and ("?" in user\_message or len(user\_message.split()) >= 4):

log\_message(f"Routing text message from {chat\_id} to AI.", component="AI\_ROUTE")

await update.message.reply\_chat\_action(action="typing")

if chat\_id not in user\_context:

user\_context[chat\_id] = [{"role": "system", "content": "You are FiLot AI, a helpful assistant knowledgeable about FiLot liquidity pools, crypto investments, and the LA! token project. Answer user questions clearly and concisely based on your training data. If you don't know the answer, say so. Keep answers focused on the product and related crypto concepts."}]

user\_context[chat\_id].append({"role": "user", "content": user\_message})

max\_context = 10

if len(user\_context[chat\_id]) > max\_context:

user\_context[chat\_id] = [user\_context[chat\_id][0]] + user\_context[chat\_id][-(max\_context - 1):]

try:

response = await asyncio.get\_event\_loop().run\_in\_executor(

executor,

lambda: deepseek\_client.chat.completions.create(

model="deepseek-chat",

messages=user\_context[chat\_id],

max\_tokens=300,

temperature=0.7

)

)

ai\_response = response.choices[0].message.content.strip()

user\_context[chat\_id].append({"role": "assistant", "content": ai\_response})

final\_message = append\_footer(append\_ask\_note(ai\_response))

await update.message.reply\_text(final\_message, parse\_mode="Markdown")

log\_message(f"Successfully processed text message via AI for chat {chat\_id}.", component="AI")

save\_user\_question(chat\_id, user\_message, ai\_response)

track\_user\_question(chat\_id, "text\_message\_ai")

except Exception as e:

log\_error(f"Error processing text message via AI for chat {chat\_id}: {str(e)}", component="AI")

await update.message.reply\_text(append\_footer("Sorry, I couldn't process that with the AI right now. You can try using specific commands like /info or /faq, or use `/ask your question`."))

else:

log\_message(f"No specific action taken for text message from {chat\_id}.", component="TELEGRAM")

pass

# --- Flask Web Server Endpoints ---

@app.route('/health')

def health\_check\_endpoint():

log\_message("Health check endpoint accessed.", component="FLASK")

node\_status\_code = 503

try:

response = \_http\_session.get(f"{NODE\_SERVICE\_URL}/health", timeout=2)

node\_status\_code = response.status\_code

except Exception:

pass

if node\_status\_code == 200:

return jsonify(status="OK", telegram\_bot="Running", node\_service="OK"), 200

else:

return jsonify(status="Partially Available", telegram\_bot="Running", node\_service=f"Status {node\_status\_code}"), 503

@app.route('/pools\_py')

def get\_pools\_via\_python():

log\_message("/pools\_py endpoint accessed.", component="FLASK")

data = fetch\_pools\_data()

if data and "fallback\_id" not in data.get("topAPR", [{}])[0].get("id", ""):

return jsonify(data), 200

else:

return jsonify(error="Failed to retrieve fresh pool data", fallback\_used=("fallback\_id" in data.get("topAPR", [{}])[0].get("id", ""))), 503

@app.route('/')

def index():

log\_message("Root endpoint '/' accessed.", component="FLASK")

uptime = str(datetime.now() - START\_TIME).split('.')[0]

return f"FiLot Telegram Bot & API Gateway. Bot Uptime: {uptime}. Status: Running. Check /health for details."

# --- Main Application Logic ---

async def init\_application():

"""Initialize and configure the bot application"""

if not TELEGRAM\_BOT\_TOKEN:

log\_error("Cannot start: TELEGRAM\_BOT\_TOKEN is not set.", component="MAIN")

return None

try:

application = Application.builder().token(TELEGRAM\_BOT\_TOKEN).build()

me = await application.bot.get\_me()

log\_message(f"Bot initialized: @{me.username}", component="INIT")

# Register command handlers with proper error handling

handlers = [

CommandHandler("start", command\_handler\_wrapper(start)),

CommandHandler("subscribe", command\_handler\_wrapper(subscribe)),

CommandHandler("unsubscribe", command\_handler\_wrapper(unsubscribe)),

CommandHandler("info", command\_handler\_wrapper(info)),

CommandHandler("simulate", command\_handler\_wrapper(simulate)),

CommandHandler("status", command\_handler\_wrapper(status)),

CommandHandler("help", command\_handler\_wrapper(help\_command)),

CommandHandler("faq", command\_handler\_wrapper(faq)),

CommandHandler("social", command\_handler\_wrapper(social)),

CommandHandler("ask", command\_handler\_wrapper(ask\_command)),

CommandHandler("ban", command\_handler\_wrapper(ban\_command)),

CommandHandler("unban", command\_handler\_wrapper(unban\_command)),

CommandHandler("joinpool", command\_handler\_wrapper(joinpool\_command)),

ChatJoinRequestHandler(callback=join\_request\_handler),

MessageHandler(filters.TEXT & ~filters.COMMAND, text\_message\_handler)

]

for handler in handlers:

application.add\_handler(handler)

log\_message("All handlers registered successfully", component="INIT")

return application

except Exception as e:

log\_error(f"Failed to initialize application: {str(e)}", component="INIT")

return None

# This is outside the if \_\_main\_\_ block

app = Flask(\_\_name\_\_)

# Initialize the bot at module level for Gunicorn workers

bot\_application = None

def initialize\_bot():

"""Initialize the bot if not already initialized"""

global bot\_application

if not bot\_application:

loop = asyncio.new\_event\_loop()

asyncio.set\_event\_loop(loop)

bot\_application = loop.run\_until\_complete(init\_application())

if bot\_application:

loop.run\_until\_complete(bot\_application.initialize())

loop.run\_until\_complete(bot\_application.start())

loop.create\_task(bot\_application.updater.start\_polling(drop\_pending\_updates=True))

loop.run\_forever()

if \_\_name\_\_ == '\_\_main\_\_':

log\_message("---- Application Starting ----", component="MAIN")

try:

# Start the bot in a separate thread

import threading

bot\_thread = threading.Thread(target=initialize\_bot, daemon=True)

bot\_thread.start()

# Run Flask for development

app.run(host='0.0.0.0', port=int(os.getenv("PORT", 5000)))

except Exception as e:

log\_error(f"Critical error: {str(e)}", component="MAIN")

sys.exit(1)