SECURE ACCESS TO ML API'S USING PYTHON

PRESENTED BY

STUDENT NAME: MANDAVA SAI SHRUTHI

COLLEGE NAME: MALLA REDDY ENGINEERING COLLEGE FOR WOMEN

DEPARTMENT: ELECTRONICS AND COMMUNICATION ENGINEERING

EMAIL ID: MANDAVASAISHRUTHI@GMAIL.COM

AICTE STUDENT ID:STU664325E7E1BB21715676647

OUTLINE

- **Problem Statement** (Should not include solution)
- **Proposed System/Solution**
- **System Development Approach** (Technology Used)
- **Algorithm & Deployment**
- **Result (Output Image)**
- **Conclusion**
- **Future Scope**
- **References**

PROBLEM STATEMENT

Many Artificial Intelligence and Machine Learning services are now accessible via APIs (e.g., OpenAI, Google Vision, Hugging Face). However, using these services securely requires managing API keys effectively . If these keys are exposed in the code or public repositories, it can lead to misuse, financial losses, and data breaches. AIML projects, which often involve sensitive data and costly API usage, are especially at risk.

PROPOSED SOLUTION

- The goal is to design and implement a secure, Python-based solution for accessing AIML APIs while protecting secret API keys.
- Core Components :Use of environment variables and .env files (python-dot env)Encrypting and decrypting keys using cryptography Accessing and consuming ML APIs securely (e.g., OpenAI's GPT, Google Vision API)Logging and error handling without leaking sensitive data
- Environment-based Key Management: Store API keys in environment variables or .env files using the python-dot env library to avoid hardcoding sensitive information.
- This ensures keys remain private even when code is shared or uploaded to GitHub .Encryption Layer: Implement secure encryption of API keys using the cryptography library. Encrypted keys can be stored safely in files or databases and decrypted at runtime only when required.
- API Integration with Error Handling: Integrate APIs such as OpenAI, Hugging Face, or Google Vision securely.
- Include robust error handling mechanisms to manage timeouts, quota limits, or expired tokens, and retry failed requests . Secure API Access: Use Python's requests module to send API requests. Authenticate using dynamic injection of secure keys and include minimal sensitive data in network requests.
- Avoid logging sensitive headers . Logging and Monitoring: Maintain logs of API usage while excluding sensitive content. Add basic monitoring functionality to detect unusual patterns, such as excessive or unauthorized access attempts . Scalable Deployment: The solution can be containerized using Docker and deployed in a cloud or on-premise environment. API keys can be injected at runtime using environment variables or integrated secret managers.
- User Interface Integration (Optional): A simple UI (web or CLI-based) can be provided to interact with the APIs and visualize output securely.

SYSTEM APPROACH

System Development Approach (Technology Used) :The tool uses Python 3.10 paired with third-party libraries for security and HTTP requests.

Technologies Used :

Python 3.8+requests, os, dotenv, cryptography AIML API (e.g., OpenAI or Hugging Face) Jupyter Notebook / VS Code Git for version control

System Requirements :

Python Environment Internet access to call ML APIs Secure local or cloud environment

ALGORITHM & DEPLOYMENT

- **Load Environment Variables**
- Use python-dotenv to load API keys securely from a .env file.
- **Encrypt Key Storage**
- Store and retrieve encrypted keys using Python's cryptography (Fernet).
- **Access ML API**
- Call an AIML API (e.g., OpenAI's ChatGPT API) securely using the protected key.

Output Handling

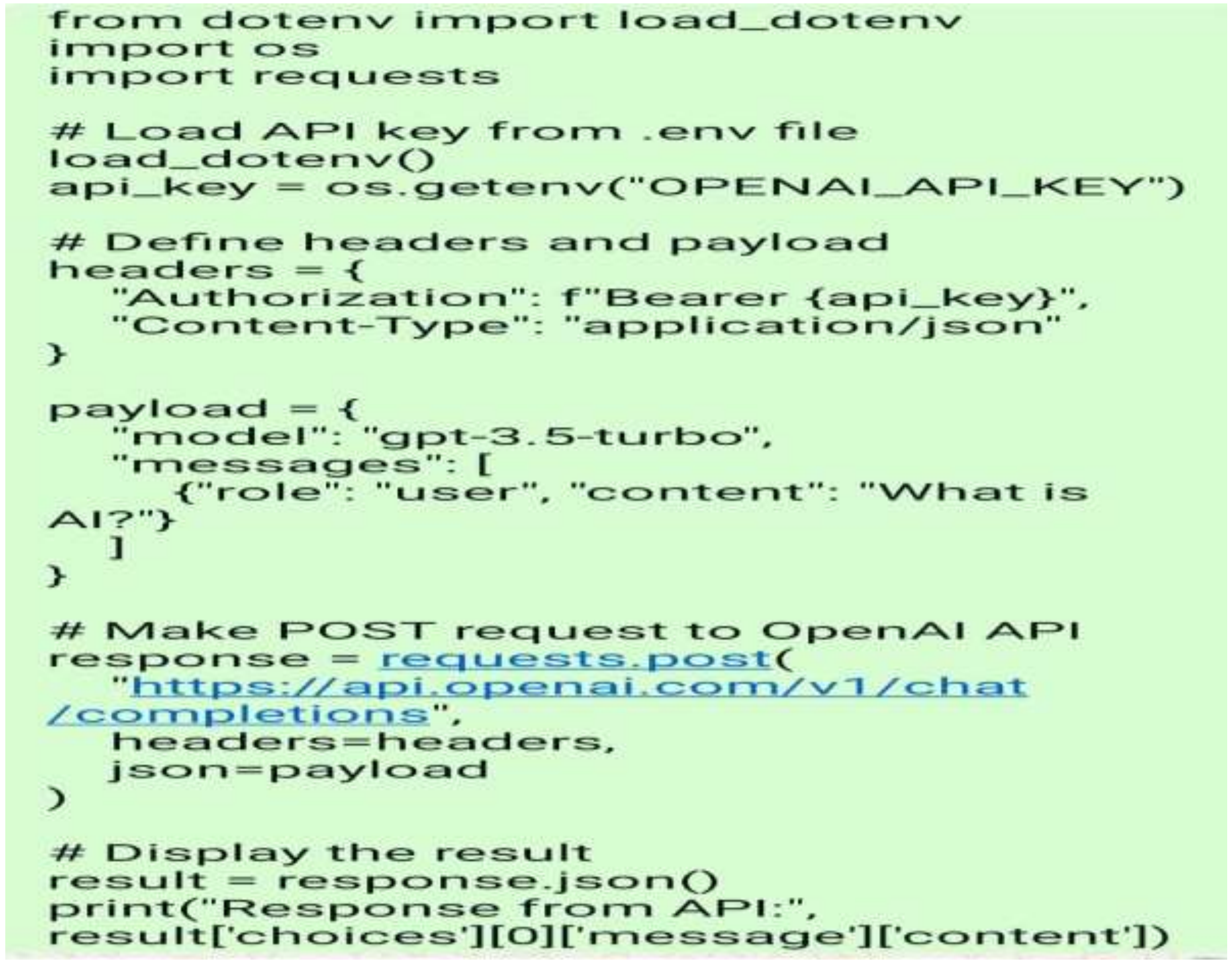
- Process and display the model's output appropriately.
- **API Key Handling:**
- Read key from .env file using dotenv.Optionally decrypt using cryptography.Fernet.
- **API Request Preparation:**
- Construct headers using secure token injection.Prepare request body including ML input data.
- **Send Request to AIML API:**
- Use requests.post() or requests.get() depending on endpoint.Handle connection timeout, status codes, and retries.

- **Parse Response**:
- Extract result from JSON (e.g., text, image classification, summary).Clean/format the output for display or downstream use.
- **Security Checks & Logging**:
- Log transaction ID, timestamp, and user action (not the key).Monitor request volume to detect misuse or quota issues.
- **UI/CLI Display**:
- Display result securely via terminal or web page Optionally cache result locally (no sensitive data).

DEPLOYMENT

- Environment Preparation:
- 1.Setup virtual environment Install required libraries (pip install -r requirements.txt)Prepare .env file or encrypted key store
- 2. Testing & Validation :Test locally with dummy or limited-access API keys Use unittest or pytest for function testing
- 3. Containerization :Write Docker file to define environment Use docker-compose for service orchestration if needed
- 4. Deployment Targets : Local machine for testing Cloud platforms (AWS EC2, Azure VM, GCP VM)CI/CD pipelines using GitHub Actions or Jenkins
- 5. Runtime Configuration: Set API keys via environment secrets (in cloud or CI/CD)Monitor logs and configure alerts on key events (failures, anomalies)
- 6. Security Practices: Apply principle of least privilege to API keys Ensure .env is in .gitignoreUse HTTPS for all API communication humans.

RESULT



```
from dotenv import load_dotenv
import os
import requests

# Load API key from .env file
load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")

# Define headers and payload
headers = {
    "Authorization": f"Bearer {api_key}",
    "Content-Type": "application/json"
}

payload = {
    "model": "gpt-3.5-turbo",
    "messages": [
        {"role": "user", "content": "What is
AI?"}
    ]
}

# Make POST request to OpenAI API
response = requests.post(
    "https://api.openai.com/v1/chat
/completions",
    headers=headers,
    json=payload
)

# Display the result
result = response.json()
print("Response from API:",
result['choices'][0]['message']['content'])
```

- **Sample Output:**
- Response from API: AI, or Artificial Intelligence, refers to the simulation of human intelligence in machines that are programmed to think and learn like humans.

CONCLUSION

- Demonstrated safe and secure usage of AIML APIs using Python.
- Applied encryption and environment management best practices.
- Prevented API key leakage—a major vulnerability in AI app development.

FUTURE SCOPE

- Integrate cloud-based secret managers (AWS Secrets Manager, GCP Secret Manager)
- Automate API key rotation and alert on abnormal API usage
- Extend to multiple ML APIs (NLP, Vision, Speech)
- Add multi-user access control and token validation system
- Introduce audit trails and anomaly detection using AI

REFERENCES

- https://12factor.net/config
- https://github.com/theskumar/python-dotenv
- https://cryptography.io
- https://platform.openai.com/docs

GitHub Link: https://github.com/saishruthi05

Thank you