Empowering Cyber Threat Intelligence with AI in practice

Motivation

- As the **number** of **cyberattacks** are **constantly increasing**, it **becomes** more and more **challenging** to **keep track** of the **threat landscape**.
- Every day, threat **analysts have to review new** CTI **reports**, data **leaks**, **vulnerabilities** and **attack vectors**.
- In addition, **skilled personnel capable** of **analyzing** all this **information** in order to **develop** appropriate **countermeasures remains** a **rare resource**.
- The **rapid evolution** of **AI**, **especially** in the field of large language models (**LLMs**), also provide fascinating **new opportunities** for **cyber security** in various domains.

This talk will focus on use cases in the field of Cyber Threat Intelligence (CTI).

Example use cases for LLMs in CTI

- Efficient **processing** and **analysis** of **leak/breachdata**: **Automatic extraction** of **relevant information** from **stolen databases**, **forum posts** or **darkweb marketplaces**.
- **Semantic analysis** of **content** to **automatically generate** threat intelligence **reports**.
- **Detection** of **patterns** and **anomalies** which human **analysts** may **potentially miss** and **recommendations** for suitable **countermeasures**.
- **Knowledge base**, in form of a **chatbot assistant aggregating information** from **various sources** and **answering** specific **questions** to analysts, for instance “Which TTPs are typical for Threat Actor abc?” or “What features does malware xyz have?”

RAG or Finetuning of LLMs? 1/2

- **Fine-tuning: Further training** of an **existing LLM** with **new data** in order to better **adapt** it to a **particular task** or **domain**. **Advantages**: **performance**, **specialized tone**, great for **summarizing information**.

- **Retrieval-Augmented Generation (RAG): Enrich** specific **information** from a **custom knowledge** database which is **not known** to the **LLM**, to **provide accurate** and **context-sensitive answers**.

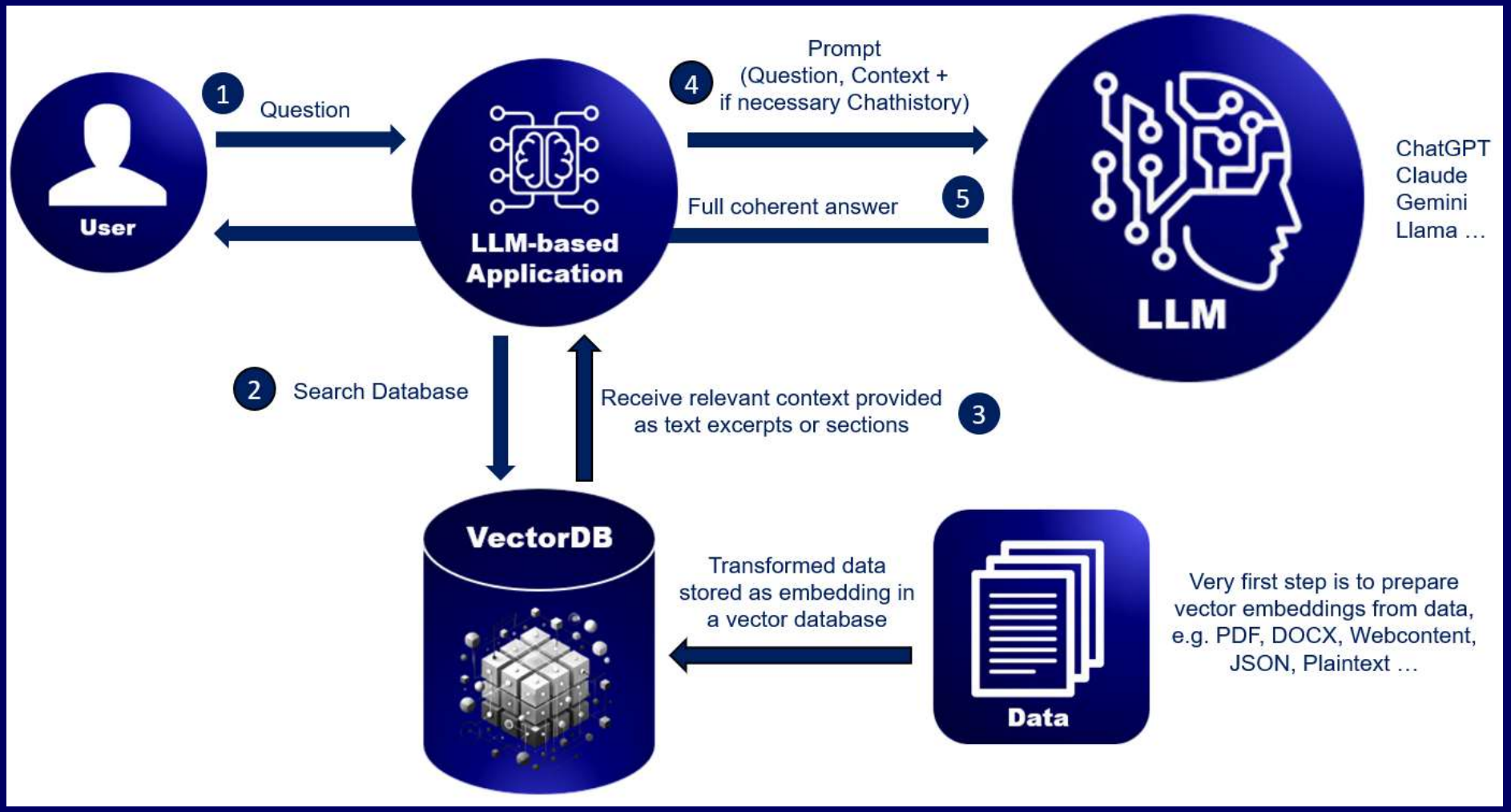
RAG or Finetuning of LLMs? 2/2

- **Fine-tuning** is a **good choice** when **specific tasks** need to be solved **with stable** and **well-defined data**. However, next to the **disadvantage** of **knowledge cut off**, **training** a **model** requires a **significant** amount of **computing resources** and **generates high costs.**

- **RAG** is particularly **useful** in **dynamic environments** where **frequent data updates** or **verifiability** of **information** is **required**. This method is characterized by **high scalability**, **flexibility** and **maintainability**, as well as **cost efficiency**.

We focus on RAG in this talk!

RAG (Retrieval Augmented Generation) illustrated



Important terms

- **Vector database**: Stores high-dimensional vectors and enables efficient **similarity searches** to **find** semantically **related content** based on numerical representations (**Embeddings**).
- To **store data** in **vectors**, **two** essential **steps** are necessary:
  - **Tokenization**: A so-called **tokenizer** breaks down **information into smaller units** (tokens) and **assigns** them an **ID**, e.g. “Hack” = 101.
  - **Embedding**: **Conversion** of the token **IDs** into high-dimensional **vectors** (using a special **embedding-model**), which **contain** information about **meaning** and **context**; words with **semantically similar** meanings receive **similar vectors** and are **close to each other**.
- **Tensor**: A flexible **data structure** tailored to **ML requirements** that can handle different dimensions of data to **perform complex mathematical operations** efficiently on **GPUs**. **Embeddings** are usually **stored as tensors**.

From text to vector illustrated

- **Example sentence:**
  “Hackback is a legitimate response to demonstrate a wolf there are more than just sheep in the enclosure.”

- **Tokenization**:
  ["Hack" → 50234, "back" → 4012, " is" → 342, " a" → 257, " legitimate" → 5678 …]
  - **Words** such as “**Hackback**” are **split** into “Hack” and “back”.
  - **Common words** such as “is”, “a”, “to” **remain** in their **basic form**.
  - **Words** such as “**legitimate**” often **remain** as a **whole** if they **occur frequently in** the **model**.
  - The **exact division** into tokens **depends on** the **tokenizer**.

- **Embedding**:
  ```
  [
      [0.321, -0.678, 0.445, ..., -0.123],  # Embedding for "Hack“
      [0.567, 0.134, -0.890, ..., 0.256],   # Embedding for "back“
      [0.213, 0.456, -0.789, ..., 0.341],   # Embedding for " is“ ….
  ]
  ```

Use Case 1: AI-generated Threat Report from a data leak

- To illustrate how AI can be used to gain insights from data leaks, I will use two popular data sources; the Conti Ransomware Gang chat leaks 2020-2022, which were published successively on the X platform shortly after the beginning of the Ukraine war. (https://x.com/ContiLeaks).
- This involves internal chat conversations between the gang members.

Hashes	Backup-Archives
936ad9f6ac2179f6a980ab8c359a7ec80bef5670	https://samples.vx-underground.org/Archive/Conti%20Leaks/Conti%20Chat%20Logs%202020.7z
52142ab7991ce99abf0d802c0b5ac6be7e38e578	https://samples.vx-underground.org/Archive/Conti%20Leaks/Conti%20Jabber%20Chat%20Logs%202021%20-%202022.7z

- Since then, multiple cybersecurity companies have published reports on the content of these leaks. These slides are therefore primarily focused on using this sample data to illustrate the steps required to develop a RAG application that can be used to gain insights from these chat logs.

Inside the chat logs

- If we **unpack** the **.7z files** we notice **148 .json files** from **Conti Chat Logs 2020.7z** and **394 .json files** from **Conti Jabber Chat Logs 2021 - 2022.7z**
- If we take a closer look at the **files**, we see that they cannot be read directly with a JSON parser because the **structure** does **not correspond** exactly to the **specifications** of a valid **JSON** array.
- In terms of **content**, there are **4 objects**: ts, from, to and body. The **latter** contains the **message** in **Russian language**.


```
{
  "ts": "2020-06-21T15:21:49.923100",
  "from": "price@q3mcco35auwcstmt.onion",
  "to": "mentos@q3mcco35auwcstmt.onion",
  "body": "как мне плагин scilla в ida pro 7.2 поставить"
}
{
  "ts": "2020-06-21T15:21:59.583023",
  "from": "price@q3mcco35auwcstmt.onion",
  "to": "mentos@q3mcco35auwcstmt.onion",
  "body": "scillahide"
}
{
  "ts": "2020-06-21T15:22:12.452131",
  "from": "price@q3mcco35auwcstmt.onion",
  "to": "mentos@q3mcco35auwcstmt.onion",
  "body": "антидебан отрубить"
}
```

Developing a Ru2En translator 1/4

- Our **first goal** is now to **translate** all the **messages** in the “Body” object to **English**.
- This step makes sense as the **large language models** are usually **optimized** for **English**.
- This can **improve** the **accuracy and reliability** of the **results** of the **semantic analysis**.

- In order to **develop** a Russian to English **translator** I used the **following packages**:
  Python 3.12.7 → https://www.python.org/downloads/release/python-3127/
  Pip Install → **torch transformers sentencepiece sacremoses chardet**

  For Torch GPU support:
  Install → **https://developer.nvidia.com/cuda-toolkit**
  python -m pip install torch --index-url https://download.pytorch.org/whl/cu124

  Tokenizer and Model for translation from ru2en:
  https://huggingface.co/Helsinki-NLP/opus-mt-ru-en

Developing a Ru2En translator 2/4

```python
import torch, re, glob, chardet, logging
from transformers import MarianMTModel, MarianTokenizer

# avoid transformer warnings, just report errors
logging.getLogger("transformers").setLevel(logging.ERROR)

# match for ts: from: to: body: entries
search_pattern = r'\{\s*"ts":\s*"(.*?)",\s*"from":\s*"(.*?)",\s*"to":\s*"(.*?)",\s*"body":\s*"((?:\\.|[^"])*)"\s*\}'

# match URLs for masking
url_pattern = r'(https?://[\w.-]+(?:/[^\s]*)?)'

directory = "./*.json"

buffer = []

# load pretrained tokenizer and model
model_name = "Helsinki-NLP/opus-mt-ru-en"
tokenizer = MarianTokenizer.from_pretrained(model_name, clean_up_tokenization_spaces=True)
model = MarianMTModel.from_pretrained(model_name)

device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)

maxlen = model.config.max_length
```

- Importing modules + loading translator tokenizer+model
- Define regex pattern for the data objects (ts, from, to, body)
- Define regex pattern for URL masking
- Init of device being used for Pytorch tensors
- Init some variables

Developing a Ru2En translator 3/4

```python
def translate_text(text, model, tokenizer, max_length=maxlen):
    # Find and mask URLs to avoid translation approaches
    urls = re.findall(url_pattern, text)
    masked_text = text
    for i, url in enumerate(urls):
        masked_text = masked_text.replace(url, f"<URL{i}>")

    # Tokenize text, use Pytorch-Tensor, splitted into chunks to avoid exceeding models max allowed length
    inputs = tokenizer(masked_text, return_tensors="pt", padding=True, truncation=True, max_length=maxlen)
    input_ids = inputs['input_ids'].to(device)
    attention_mask = inputs['attention_mask'].to(device)
    translated_text = []

    for i in range(0, input_ids.size(1), maxlen):
        input_chunk = input_ids[:, i:i + maxlen]
        # To deal with a batch of input ids varying in length, we are padding all sequence
        # to the same length and use the attention mask tensor for identification
        attention_chunk = attention_mask[:, i:i + maxlen]
        # Input gets translated, output is a tensor, with model translated tokens
        translated_chunk = model.generate(input_ids=input_chunk, attention_mask=attention_chunk)
        # Build translated text from tokens, no special tokens
        translated_text.append(tokenizer.decode(translated_chunk[0], skip_special_tokens=True))

    translated_text = " ".join(translated_text)

    # Replace masked URLs back into the text
    for i, url in enumerate(urls):
        translated_text = translated_text.replace(f"<URL{i}>", url)

    return translated_text
```

When using Pytorch tensors for tokenization and translation text needs to be splitted in chunks depending on the models allowed max length

As input IDs can vary, we need padding and use attention masking

Skip special model control tokens in the translated text

Mask all kind of URLs to avoid translation approaches by the model

Developing a Ru2En translator 4/4

```python
for file in glob.glob(directory):
    with open(file, 'rb') as rf:
        result = chardet.detect(rf.read())
    print(f"\nProcessing => {file}")
    with open(file, "r", encoding=result['encoding'], errors="ignore") as file:
        matches = re.findall(search_pattern, file.read())
        for ts, from_addr, to_addr, body in matches:
            buffer.append(f"ts: {ts}")
            buffer.append(f"from: {from_addr}")
            buffer.append(f"to: {to_addr}")
            translated_body = translate_text(body, model, tokenizer, maxlen)
            buffer.append(f"body: {translated_body}\n")

full_translated_text = "\n".join(buffer)

with open("full_translated.txt", "w", encoding="utf-8") as f:
    f.write(full_translated_text)
```

- Load all files in directory
- Detect char encoding
- Parse objects, in case of „body“-object translate text from ru2en
- Write a single file with all entries

Example of translated text

```
ts: 2020-06-21T15:21:49.923100
from: price@q3mcco35auwcstmt.onion
to: mentos@q3mcco35auwcstmt.onion
body: how do i put the scilla plugin in ida pro 7.2?
```

NOTE: The **translated texts** are **not** that **bad**, but for **better quality** start playing with **other models or** use a professional **cloud service** like **Deepl** for the **best translation** u can actually get.

Translating with Deepl API

```python
import os, deepl, glob, chardet, re

# match for ts: from: to: body: entries
search_pattern = r'\{\s*"ts":\s*"(.*?)",\s*"from":\s*"(.*?)",\s*"to":\s*"(.*?)",\s*"body":\s*"((?:\\.|[^"])*)"\s*\}'

directory = "./*.json"

buffer = []

def translate_text(text, translator):
    try:
        translated_text = translator.translate_text(text, source_lang="RU", target_lang="EN-US")
        return translated_text.text
    except deepl.exceptions.DeepLException as e:
        print(f"Error during translation: {e}")
        return "<bad request>"

translator = deepl.Translator(os.environ["DEEPL_API_KEY"])

for file in glob.glob(directory):
    with open(file, 'rb') as rf:
        result = chardet.detect(rf.read())
    print(f"\nProcessing => {file}")
    with open(file, "r", encoding=result['encoding'], errors="ignore") as file:
        matches = re.findall(search_pattern, file.read())
        for ts, from_addr, to_addr, body in matches:
            buffer.append(f"ts: {ts}")
            buffer.append(f"from: {from_addr}")
            buffer.append(f"to: {to_addr}")
            translated_body = translate_text(body, translator)
            buffer.append(f"body: {translated_body}\n")

full_translated_text = "\n".join(buffer)

with open("full_translated.txt", "w", encoding="utf-8") as f:
    f.write(full_translated_text)
```

Adjusted translator code using the Deepl-API

For more information check:
https://developers.deepl.com/docs

More important terms 1/3

- **LangChain**: **Enables AI applications** to be **created** and **orchestrated** by **integrating language models**, **data sources** and **tools** such as **storage**, **APIs** and **knowledge graphs**.
- **OpenAI Development Platform**: P**rovides APIs** and **tools** to **integrate AI models** such as GPT and DALL-E **into applications** to **generate text**, **create images** and **analyze data**.
- **FAISS (Facebook AI Similarity Search)**: **Library** for **searching** and **comparing vector data** to **find similar entries**. This is particular **helpful** in applications such as **semantic search**, **recommendation systems** and **clustering**.
- **Temperature**: This parameter influences how creative or conservative the answers of AI generated text is.
  - A **lower number** (e.g. 0.2) → **answers** are more **focused** and **predictable** (model chooses most likely answer).
  - A **higher number** (e.g. 1.0) → **answers** are more **creative**, and **less predictable** (model chooses less likely answer more often).
- **Top_k**: **Determines how many** of the **most probable** next **words** should be **considered**.
  - **Lower number** (e.g. k=1) → **Most probable word** is always **taken**.
  - **Higher number** (e.g. k=50) → **Model** has **more freedom** to be **creative**.

More important terms 2/3

- **When querying information from** your **own database**, **context** information is **transmitted to** the **LLM** (**next to** the **question and** other **prompting information**).
  - Due to **token limitations** (https://platform.openai.com/docs/models/), the **transmitted data cannot** be of **arbitrary length**, though **newer models** provide **huge token context** windows.
- **Chunking**: To **solve** this **problem**, **data** is **cut** into **meaningful sections**.
- As more **complex contexts can** be **distributed across** several **chunks**, a **chunk-overlap** is defined, **in addition** to a **chunk size** to **ensure** the **end of** the **previous chunk** is always **present at** the **beginning** of the **next chunk**, to establish a connection between the sections, which **improves coherence** during processing and **minimizing context loss** between chunks.
- **Example → RecursiveCharacterTextSplitter**: Takes a text and splits it into pieces based on a specific pattern using definable characters, e.g. [“\n\n”, “\n”, “ ‘, ’”]. The first step is a division using “\n\n”. If the chunk is then still larger than the chunk size, the next character “\n” follows and so on until the specified size is reached.
  - For **more on Textsplitters** I suggest reading → https://medium.com/@263akash/different-levels-of-text-splitting-chunking-ce9da78570d5

More important terms 3/3

- **Prompting**: Refers to the **creation of inputs sent to** a **LLM** to **obtain** a **desired response** or **action**. The **prompt** is a **combination** of **text and instructions** that **control** the **behavior** and **response** of the **model**.
- **Key components:**
- **Role** → Defines **who** or **what** the **model should be**, **helping** the **LLM** to **understand** the **perspective** or **context** from **which** it **should respond**. This also **influences** the **tone**, **choice** of **words** and **approach of** the **model**.
  - Examples: “You are a threat intelligence analyst.” or “You are a CISO.”
- **Context** → **Includes relevant information** the **model needs** to **generate** a suitable **answer**.
  - Examples: “We are talking about Russian cybercrime Gangs.” or “Your task is to analyze the provided data and generate a detailed threat intelligence report.”
- **Response requirements** → **Defines** how the **response should be prepared**, e.g. format, length, style etc.
  - Example: “The report needs to be structured text in Markdown format, including specific sections such as headings, bulleted lists, and tables for threat details. Do NOT add information that is not explicitly provided in the context.”

Developing the CONTI Leaks RAG bot 1/4

```python
import chardet, tiktoken
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# Calculate number of tokens of a given text based on the tokenization of the "gpt-4o" model
def CalcNumTokens(text):
    encoding = tiktoken.encoding_for_model("gpt-4o")
    tokens = encoding.encode(text, disallowed_special=())
    return len(tokens)

file=".\\conti2021+22-translated.txt"

# Detect file encoding
with open(file, 'rb') as rf:
    result = chardet.detect(rf.read())

# Read file with the specified encoding
with open(file, 'r', encoding=result['encoding'], errors="ignore") as file:
    data=file.read()

# Split text into chunks using the RecursiveCharacterTextSplitter
tsplit = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=CalcNumTokens,
    separators=['\n\n', '\n', ' ', '']
)
chunks = tsplit.split_text(data)
```

Importing modules

Read source file

Use RecursiveCharacterTextSplitter for chunking

```
>>> chunks[0]
"ts: 2021-01-29T00:06:46.929363\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stern@q3mcco35auwcstmt.onion\nbody: don't forget about the bits, kosh above, I'm going to bed)\n\nts:
2021-01-29T04:04:39.308133\nfrom: mango@q3mcco35auwcstmt.onion\nto: stern@q3mcco35auwcstmt.onion\nbody:
hello\n\nts: 2021-01-29T04:04:43.474243\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stern@q3mcco35auwcstmt.onion\nbody: there's not enough bits for everything.\n\nts:
2021-01-29T04:32:02.648304\nfrom: price@q3mcco35auwcstmt.onion\nto: green@q3mcco35auwcstmt.onion\nbody:
hi!!!\n\nts: 2021-01-29T04:32:16.858754\nfrom: price@q3mcco35auwcstmt.onion\nto:
green@q3mcco35auwcstmt.onion\nbody: Did they change the gaskets again??? No connection!\n\nts:
2021-01-29T04:33:01.808125\nfrom: green@q3mcco35auwcstmt.onion\nto: price@q3mcco35auwcstmt.onion\nbody:
Hi\n\nts: 2021-01-29T05:04:52.370538\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stakan@q3mcco35auwcstmt.onion\nbody: Hi, I'm waiting for the kosh and the amount in the btz.\n\nts:
2021-01-29T06:34:42.811135\nfrom: stakan@q3mcco35auwcstmt.onion\nto:
mango@q3mcco35auwcstmt.onion\nbody: hello\n\nts: 2021-01-29T06:39:46.323651\nfrom:
stakan@q3mcco35auwcstmt.onion\nto: mango@q3mcco35auwcstmt.onion\nbody:
bc1qy2083z665ux68zda3tfuh5xed2493uaj8whdwv - 0.02260047\n\nts: 2021-01-29T06:48:51.062571\nfrom:
mango@q3mcco35auwcstmt.onion\nto: stakan@q3mcco35auwcstmt.onion\nbody: moment\n\nts:
2021-01-29T06:52:12.583011\nfrom: many@q3mcco35auwcstmt.onion\nto: mango@q3mcco35auwcstmt.onion\nbody:
answer there.\n\nts: 2021-01-29T06:53:08.150417\nfrom: mango@q3mcco35auwcstmt.onion\nto:
many@q3mcco35auwcstmt.onion\nbody: answer there.\n\nts: 2021-01-29T07:38:39.880309\nfrom:
baget@q3mcco35auwcstmt.onion\nto: globus@q3mcco35auwcstmt.onion\nbody: Hi.\n\nts:
2021-01-29T07:52:56.702134\nfrom: mango@q3mcco35auwcstmt.onion\nto:
vampire@q3mcco35auwcstmt.onion\nbody: Hey, bro.\n\nts: 2021-01-29T07:53:02.659462\nfrom:
mango@q3mcco35auwcstmt.onion\nto: vampire@q3mcco35auwcstmt.onion\nbody: I'm waiting for my paycheck today
and the amount.\n\nts: 2021-01-29T08:15:57.669241\nfrom: vampire@q3mcco35auwcstmt.onion\nto:
mango@q3mcco35auwcstmt.onion\nbody: hello"
```

Developing the CONTI Leaks RAG bot 2/4

```
# Convert text chunks into LangChain Document objects
DocumentObjects = []
for i, chunk in enumerate(chunks):
    DocumentObjects.append(Document(page_content=chunk, metadata={'chunk': i}))
```

```
>>> DocumentObjects[0]
Document(metadata={'chunk': 0}, page_content="ts:
2021-01-29T00:06:46.929363\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stern@q3mcco35auwcstmt.onion\nbody: don't forget about the bits, kosh above,
I'm going to bed)\n\nts: 2021-01-29T04:04:39.308133\nfrom:
mango@q3mcco35auwcstmt.onion\nto: stern@q3mcco35auwcstmt.onion\nbody:
hello\n\nts: 2021-01-29T04:04:43.474243\nfrom:
mango@q3mcco35auwcstmt.onion\nto: stern@q3mcco35auwcstmt.onion\nbody:
there's not enough bits for everything.\n\nts: 2021-01-29T04:32:02.648304\nfrom:
price@q3mcco35auwcstmt.onion\nto: green@q3mcco35auwcstmt.onion\nbody:
hi!!!\n\nts: 2021-01-29T04:32:16.858754\nfrom:
price@q3mcco35auwcstmt.onion\nto: green@q3mcco35auwcstmt.onion\nbody: Did
they change the gaskets again??? No connection!\n\nts:
2021-01-29T04:33:01.808125\nfrom: green@q3mcco35auwcstmt.onion\nto:
price@q3mcco35auwcstmt.onion\nbody: Hi\n\nts:
2021-01-29T05:04:52.370538\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stakan@q3mcco35auwcstmt.onion\nbody: Hi, I'm waiting for the kosh and the
amount in the btz.\n\nts: 2021-01-29T06:34:42.811135\nfrom:
stakan@q3mcco35auwcstmt.onion\nto: mango@q3mcco35auwcstmt.onion\nbody:
hello\n\nts: 2021-01-29T06:39:46.323651\nfrom:
stakan@q3mcco35auwcstmt.onion\nto: mango@q3mcco35auwcstmt.onion\nbody:
bc1qy2083z665ux68zda3tfuh5xed2493uaj8whdwv - 0.02260047\n\nts:
2021-01-29T06:48:51.062571\nfrom: mango@q3mcco35auwcstmt.onion\nto:
stakan@q3mcco35auwcstmt.onion\nbody: moment\n\nts:
```

Document objects can store metadata (e.g. chunk number, source, category etc.) alongside page content.

FAISS supports a metadata-based search.

Retrieved chunks are not examined isolated, instead they are evaluated in connection with the request and the metadata.

The system “understands” that certain metadata influences the relevance of a chunk for the query.

Context-aware retrieval improves searches and generates higher quality answers.

Developing the CONTI Leaks RAG bot 3/4

```python
# Create a FAISS vector database from DocumentObjects
embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
vectordb = FAISS.from_documents(DocumentObjects, embeddings)

# LLM and Prompt Setup
llm = ChatOpenAI(temperature=0.2, model_name="gpt-4o")

prompt_template = """
You are a highly specialized Cyber Threat Intelligence Analyst with 20 years of experience in analyzing cybercrime operations.

The provided data are leaked chat logs from a cybercrime gang namend CONTI, involved in ransomware activities.
Your task is to analyze the data provided very thoroughly and to prepare the insights gained from it.
Do NOT add information that is not explicitly provided in the context.

Context: {context}

Question: {question}
"""

prompt = PromptTemplate(template=prompt_template, input_variables=["context", "question"])

# Build a RAG-Chain
retriever = vectordb.as_retriever(search_kwargs={
    "fetch_k": 20, # fetch the top 20 relevant chunks from the vector database
    "k": 10, # select the top 10 chunks from the fetched ones
    "maximal_marginal_relevance": True # additionally avoid chunks with redundant information
})

ragchain = (
    {"context": retriever, "question": RunnablePassthrough()} # get context from retriever and pass through question
    | prompt
    | llm
    | StrOutputParser() # convert output to string
)
```

- Use OpenAI's large text embedding for creating a FAISS based vector database from document objects
- Once the vector database has been created it can be stored using vectordb.save_local() and loaded via FAISS.load_local()
- Setup the LLM, configure parameters (Temperature and model) and prompt template
- Create a retriever for the vector database and configure parameters (fetch_k, k, mmr)
- Setup the RAG-chain

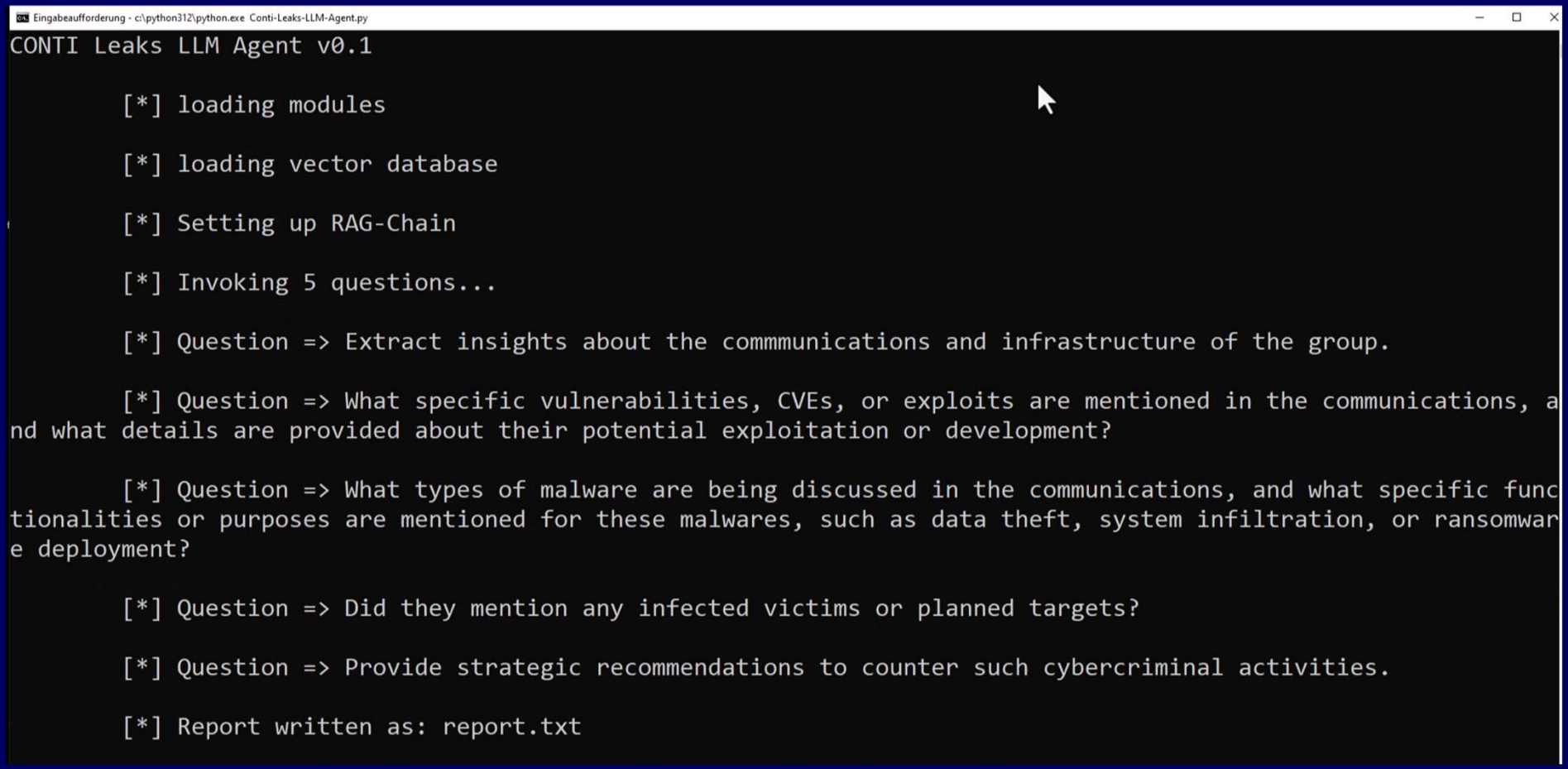
Developing the CONTI Leaks RAG bot 4/4

```python
# Definition of questions and report generation
questions = [
    "Extract insights about the communications and infrastructure of the group.",
    "What specific vulnerabilities, CVEs, or exploits are mentioned in the communications, and what details are provided about their potential exploitation or development?",
    "What types of malware are being discussed in the communications, and what specific functionalities or purposes are mentioned for these malwares, such as data theft, system infiltration, or ransomware deployment?",
    "Did they mention any infected victims or planned targets?",
    "Provide strategic recommendations to counter such cybercriminal activities."
]

report_sections = []
for question in questions:
    print(f"\n\n[*] Question => {question}\n")
    result = ragchain.invoke(question)
    report_sections.append(result)

full_report = "\n\n".join(f"### {questions[i]}\n\n{section}" for i, section in enumerate(report_sections))

with open("report.txt", "w", encoding="utf-8") as f:
    f.write(full_report)
```

Define a set of questions and process them by invoking the RAG-chain for every question

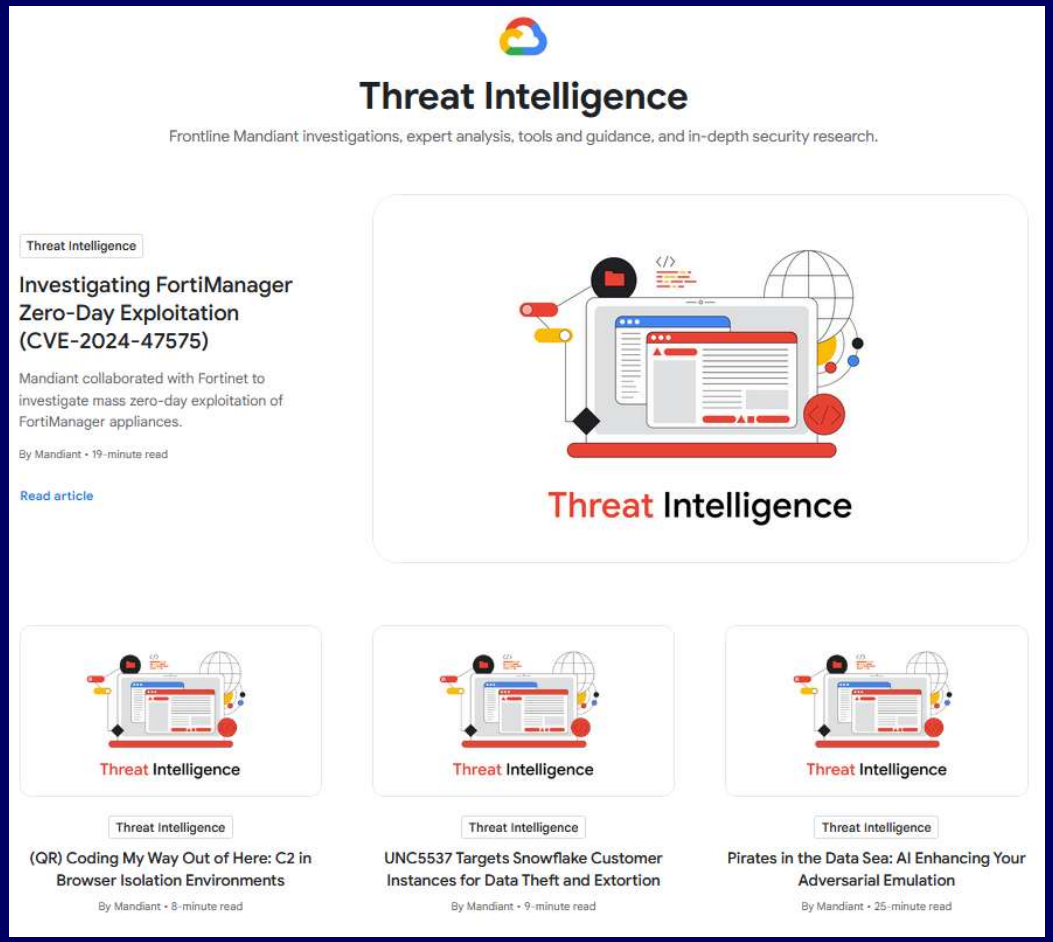
Join all results to a report and store it as file.

The Conti Leaks LLM Agent in action



Video-Link: https://github.com/fboldewin/Empowering-CTI-with-AI-in-practice/raw/refs/heads/main/CONTI Leaks LLM Agent.mp4

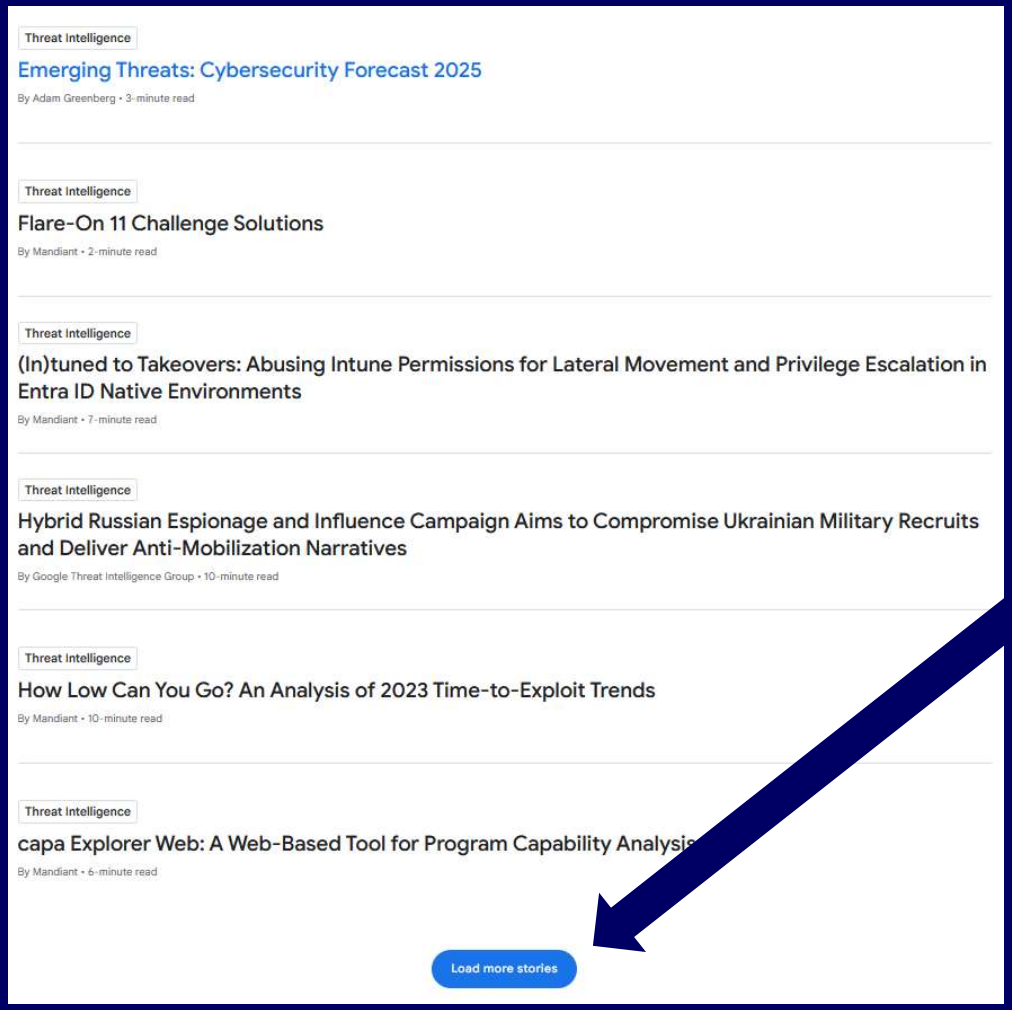
Use Case 2: CTI-Chatbot with memory + WebUI



The second use case shows how to **develop** a **CTI chatbot** with which a **user** can **interact**, e.g. to **ask specific questions** about threat actors etc.
The chatbot should be able to **include** the **conversation in** the **context** in order to **improve** the **response quality**.

For the **showcase**, hundreds of freely available **threat intelligence articles** from the **Mandiant blog** serve as a **source**.
https://cloud.google.com/blog/topics/threat-intelligence

Developing a CTI-Chatbot 1/9



First, we need all the links to the respective CTI articles belonging to the blog.

Unfortunately, it seems that there is no sitemap providing all links.

Furthermore, not all available articles appear at a glance when loading.

If you scroll to the bottom of the page, you will find the note: “Load more stories”.

To get every link without manual clicking the cross-browser automation framework “Playwright” comes in handy.

Playwright → **https://playwright.dev**

Developing a CTI-Chatbot 2/9

```python
from playwright.sync_api import sync_playwright
import time
from bs4 import BeautifulSoup

url = 'https://cloud.google.com/blog/topics/threat-intelligence/'

print("\nGetting all blogs links from: %s\nPlease wait, this takes some time...\n" % url)

with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()
    page.goto(url)

    while True:
        try:
            page.evaluate("window.scrollTo(0, document.body.scrollHeight);") # scroll to page end
            time.sleep(1) # wait a second to get the contents loaded

            # wait until available in DOM
            load_more_button = page.wait_for_selector("button:has-text('Load more stories')", timeout=10000)

            if load_more_button:
                is_visible = load_more_button.is_visible() # Is button visible?
                if is_visible:
                    load_more_button.click() # click button
                else:
                    print("All Links discovered")
                    break
            else:
                print('No 'Load more stories'-Button found! Check Website manually if its design has been changed.')
                break
        except:
            break

    content = page.content()
    browser.close()
    soup = BeautifulSoup(content, 'html.parser')
    unique_urls = list({link['href'] for link in soup.find_all('a', href=True) if link['href'].startswith(url)})
```

Load Playwright and BeautifulSoup modules for scraping.

Launch Chrome in headless mode and get Mandiant CTI Blog main page.

Scroll and click repeatedly the “Load more stories” button, until all blog links are visible on the page.

Parse HTML content of the fully loaded page with BeautifulSoup and extract all unique links starting with the base URL.

Developing a CTI-Chatbot 3/9

```python
documents = []

for url in unique_urls:
    j = 0
    try:
        response = requests.get(url)
        response.raise_for_status()
    except requests.RequestException as e:
        print(f"Error requesting URL {url}: {e}")
        continue
    soup = BeautifulSoup(response.content, 'html.parser')
    # get all tables and extract header and data rows
    tables = soup.find_all("table")
    for table in tables:
        rows = table.find_all("tr")
        if len(rows) == 0:
            continue
        headers = []
        if rows[0].find_all("th"):
            headers = [header.get_text(strip=True) for header in rows[0].find_all("th")]
            data_rows = rows[1:]
        else:
            headers = [cell.get_text(strip=True) for cell in rows[0].find_all("td")]
            data_rows = rows[1:] if len(rows) > 1 else []
        if not headers:
            continue
        table_data = []
        for row in data_rows:
            cells = row.find_all("td")
            if len(cells) != len(headers):
                continue
            row_data = {headers[i]: cells[i].get_text(strip=True) for i in range(len(cells))}
            table_data.append(row_data)
        table_doc_content = f"Table: {headers}\n" + "\n".join([str(row) for row in table_data])
        # assign metadata items to document object to table
        documents.append(Document(page_content=table_doc_content, metadata={'chunk': j, 'type': 'table', 'source': url, 'title': soup.title.text}))
        j = j + 1
    start_cut = "Get started for free Threat Intelligence" # start marker
    end_cut = "Posted in" # end marker
    page_text = soup.get_text()
    if start_cut in page_text and end_cut in page_text: # cut out the article text
        relevant_text = page_text.split(start_cut, 1)[1].split(end_cut, 1)[0].strip()
    else:
        relevant_text = page_text.strip()
    tsplit = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100, length_function=tiktoken_len, separators=['\n\n', '\n', ' ', ''])
    chunks = tsplit.split_text(relevant_text)
    for i, chunk in enumerate(chunks, start=j):
        # assign metadata items to document object to text
        documents.append(Document(page_content=chunk, metadata={'chunk': i, 'type': 'text', 'source': url, 'title': soup.title.text}))

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
vectordb = FAISS.from_documents(documents, embeddings) # create a FAISS vector database
```

Iterate over all blog links and retrieve website content using requests and BeautifulSoup module.

Search for tables in the HTML code and extract header and data rows and save as document objects to preserve the context for later searches, e.g. a table with hashes and filenames.

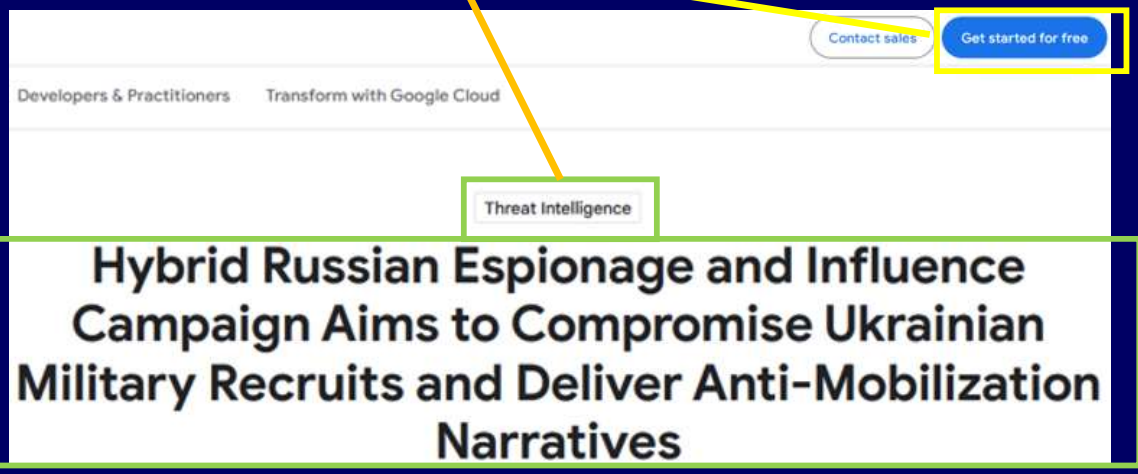
Based on start and end markers, only the texts associated with the article are cut out and then split into chunks.

Assign metadata to all tables and texts as part of their document objects:
chunk index, type (table or text), source-url and title of the web page.

Create a FAISS vector database from all chunked articles.

Developing a CTI-Chatbot 4/9

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> url = "https://cloud.google.com/blog/topics/threat-intelligence/russian-espionage-influence-ukrainian-military-recruits-anti-mobilization-narratives"
>>> response = requests.get(url)
>>> soup = BeautifulSoup(response.content, 'html.parser')

>>> soup.get_text()
Hybrid Russian Espionage and Influence Campaign Aims to Compromise Ukrainian Military Recruits and Deliver Anti-Mobilization Narratives | Google
Cloud BlogJump to ContentCloudBlogContact sales Get started for free CloudBlogSolutions & technologyAI & Machine LearningAPI
ManagementApplication DevelopmentApplication ModernizationChrome EnterpriseComputeContainers & KubernetesData AnalyticsDatabasesDevOps &
SREMaps & GeospatialSecuritySecurity & IdentityThreat IntelligenceInfrastructureInfrastructure ModernizationNetworkingProductivity &
CollaborationSAP on Google CloudStorage & Data TransferSustainabilityEcosystemIT LeadersIndustriesFinancial ServicesHealthcare & Life
SciencesManufacturingMedia & EntertainmentPublic SectorRetailSupply ChainTelecommunicationsPartnersStartups & SMBTraining & CertificationsInside
Google CloudGoogle Cloud Next & EventsGoogle Maps PlatformGoogle WorkspaceDevelopers & PractitionersTransform with Google CloudContact sales
Get started for free Threat IntelligenceHybrid Russian Espionage and Influence Campaign Aims to Compromise Ukrainian Military Recruits and Deliver
Anti-Mobilization NarrativesOctober 28, 2024Google Threat Intelligence Group In September 2024, Google Threat Intelligence Group (consisting of
Google's Threat Analysis Group (TAG) and Mandiant) discovered UNC5812, a suspected Russian hybrid espionage and influence operation, delivering
```



Start marker after which the article starts → "Get started for free Threat Intelligence" is consistent across all blog articles. ☺

Developing a CTI-Chatbot 5/9

Indicators of Compromise\nFor a more comprehensive set of UNC5812 indicators of compromise, a Google Threat Intelligence Collection is available for registered users.\n\n\n\n\n\n\n\n\n\n\n\n\nIndicators of Compromise\n\nContext\n\n\n\ncivildefense[.]com[.]ua\n\nUNC5812 landing page\n\n\n\nnt[.]me/civildefense_com_ua\n\nUNC5812 Telegram channel\n\n\n\nnt[.]me/UAcivildefenseUA\n\nUNC5812 Telegram account\n\n\n\n\ne98ee33466a270edc47fdd9faf67d82e\n\nSUNSPINNER decoy\n\n\n\nh315225216.nichost[.]ru\n\nResolver used in SUNSPINNER decoy\n\n\n\nfu-laravel.onrender[.]com\n\nHostname used in SUNSPINNER decoy\n\n\n\n206.71.149[.]194\n\nC2 used to resolve distribution URLs\n\n\n\n185.169.107[.]44\n\nOpen directory used for malware distribution\n\n\n\nd36d303d2954cb4309d34c613747ce58\n\nPronsis Loader dropper\n\n\n\nb3cf993d918c2c61c7138b4b8a98b6bf\n\nPURESTEALER\n\n\n\n31cdae71f21e1fad7581b5f305a9d185\n\nCRAXSRAT\n\n\n\naab597cdc5bc02f6c9d0d36ddeb7e624\n\nCRAXSRAT w/ SUNSPINNER decoy\xa0\n\n\n\n\n\n\n\n\n\n\n\nPosted inThreat IntelligenceRelated articlesThreat IntelligenceXRefer: The Gemini-Assisted Binary NavigatorBy Mandiant • 26-minute read

b3cf993d918c2c61c7138b4b8a98b6bf	PURESTEALER
31cdae71f21e1fad7581b5f305a9d185	CRAXSRAT
aab597cdc5bc02f6c9d0d36ddeb7e624	CRAXSRAT w/ SUNSPINNER decoy

Posted in Threat Intelligence

Related articles

End marker after which the article end → “Posted in” is consistent across all blog articles as well. ☺

Developing a CTI-Chatbot 6/9


```
Document(metadata={'chunk': 5, 'type': 'table', 'source': 'https://cloud.google.com/blog/topics/threat-intelligence/unc2891-overview/',
'title': 'Have Your Cake and Eat it Too? An Overview of UNC2891 | Mandiant | Google Cloud Blog'},
page_content="Table: ['Malware Family', 'MD5', 'SHA1', 'SHA256']\n{'Malware Family': 'STEELCORGI', 'MD5':
'e5791e4d2b479ff1dfee983ca6221a53', 'SHA1': 'e55514b83135c5804786fa6056c88988ea70e360', 'SHA256':
'95964d669250f0ed161409b93f7a131bfa03ea302575d555d91ab5869391c278'}\n{'Malware Family': 'STEELCORGI', 'MD5':
'0845835e18a3ed4057498250d30a11b1', 'SHA1': 'c28366c3f29226cb2677d391d41e83f9c690caf7', 'SHA256':
'7d587a5f6f36a74dcfbcbaecb2b0547fdf1ecdb034341f4cc7ae489f5b57a11d'}\n{'Malware Family': 'STEELCORGI', 'MD5':
'd985de52b69b60aa08893185029bcb31', 'SHA1': 'a3e75e2f700e449ebb62962b28b7c230790dc25d', 'SHA256':
'cd06246aff527263e409dd779b517157882a1f5f74a84ad78b3b0c44d306bfc3'}\n{'Malware Family': 'TINYSHELL', 'MD5':
'4ff6647c44b0417c80974b806b1fbcc3', 'SHA1': 'fa36f10407ed5a6858bd1475d88dd35927492f52', 'SHA256':
'55397addbea8e5efb8e6493f3bd1e99f9742ff4cfe0f0d3da7e92067904b5194'}\n{'Malware Family': 'TINYSHELL', 'MD5':
'13f6601567523e6a37f131ef2ac4390b', 'SHA1': '4228d71c042d08840089895bfa6bd594b5299a89', 'SHA256':
'24f459a2752175449939037d6a1da09cac0e414020ce9c48bcef47ec96e3587b'}\n{'Malware Family': 'TINYSHELL', 'MD5':
'4e9967558cd042cac8b12f378db14259', 'SHA1': '018bfe5b9f34108424dd63365a14ab005e249fdd', 'SHA256':
'5f46a25473b9dda834519093c66cced0e3630378c2a953ebd83f90f3777f2e19'}\n{'Malware Family': 'STEELHOUND', 'MD5':
'a4617c9a4bde94e867f063c28d763766', 'SHA1': '097d3a15510c48cdb738344bdf00082e546827e8', 'SHA256':
'161a2832baba6ff6f9f1b52ed8facfa1197cfc7947fe58152b3617a258cf52b0'}")
```


Examples of chunks with metadata type → table or text + source and title

page_content contains a table or text chunk of an article


```
Document(metadata={'chunk': 6, 'type': 'text', 'source': 'https://cloud.google.com/blog/topics/threat-intelligence/unc2891-overview/',
'title': 'Have Your Cake and Eat it Too? An Overview of UNC2891 | Mandiant | Google Cloud Blog'},
page_content='Have Your Cake and Eat it Too? An Overview of UNC2891March 16, 2022Mandiant Written by: Mathew Potaczek, Takahiro
Sugiyama, Logeswaran Nadarajan, Yu Nakamura, Josh Homan, Martin Co, Sylvain Hirsch\nThe Mandiant Advanced Practices team
previously published a threat research blog post that provided an overview of UNC1945 operations where the actor compromised
managed services providers to gain access to targets in the financial and professional consulting industries.\nSince that time, Mandiant
has investigated and attributed several intrusions to a threat cluster we believe has a nexus to this actor, currently being tracked as
UNC2891.'
```

Developing a CTI-Chatbot 7/9

```python
import streamlit as st
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory

# Set app title in browser window
st.set_page_config(page_title="Threat Intelligence LLM Chatbot v0.1", layout="centered")

st.markdown("""
    <style>
    .title {
        font-size:40px !important;
        font-family: Arial, sans-serif;
        text-align: center;
        font-weight: bold;
        color: white;
    }
    .logo {
        display: block;
        margin-left: auto;
        margin-right: auto;
    }
    }
    </style>
    """, unsafe_allow_html=True)

# Display the title at the center of the page
st.markdown('<p class="title">Threat Intelligence LLM Chatbot v0.1</p>', unsafe_allow_html=True)

# Display and center logo.png
st.image('logo.png', caption='', width=700, output_format='PNG', clamp=False)
```

We use Streamlit as WebUI, a library to quickly and easily create interactive web applications for data analysis and machine learning.

Load different langchain modules we need for our CTI-Chatbot.

Setup browser and page title, as well as our logo for the App.

Developing a CTI-Chatbot 8/9

```python
def load_embeddings():
    embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
    return embeddings

def load_vectorstore(embeddings):
    vectordb = FAISS.load_local("./data/FAISS_VECTORDB_MANDIANT_CTI_ARTICLES_WITH_TABLES", embeddings, allow_dangerous_deserialization=True)
    return vectordb

def initialize_chain():
    embeddings = load_embeddings()
    vectordb = load_vectorstore(embeddings)
    retriever = vectordb.as_retriever(search_kwargs={"k": 8, "fetch_k": 20, "maximal_marginal_relevance": True})

    prompt_template = """
        You are a highly experienced Cyber Threat Intelligence Analyst,
        specialized in the analysis of operations conducted by cybercrime gangs and state-sponsored threat actors.
        Your task is to analyze the data for interesting cyber threat information.

        Chat History: {chat_history}

        Context: {context}

        Based on the provided context, answer the following question thoroughly and precisely:
        Question: {question}

        Requirements for the response:
        - If the provided context does not include enough information, explicitly state: "The information is insufficient to answer this question."
        - Do not add information that is not explicitly provided in the context.
    """

    PROMPT = PromptTemplate(template=prompt_template, input_variables=["chat_history", "context", "question"])
    chain_type_kwargs = { "prompt" : PROMPT }

    llm = ChatOpenAI(temperature=0.2, model_name="gpt-4o")

    # Initialize memory to save the conversation history
    memory = ConversationBufferMemory(memory_key='chat_history', return_messages=True)

    # ConversationalRetrievalChain uses history to combine it with retrieved information
    qa_chain = ConversationalRetrievalChain.from_llm(
        llm, retriever, memory=memory, combine_docs_chain_kwargs=chain_type_kwargs
    )
    return qa_chain
```

Init OpenAI's large text embedding and load our prepared FAISS vector db.

Create a retriever for the vector database and configure parameters (fetch, k, mmr).

Setup the RAG-chain, including prompt, temperature and model selection, as well as init memory to save the conversation history and combining history with retrieved information.

The memory key „chat_history" is defined in the prompt.

Developing a CTI-Chatbot 9/9

```python
if 'qa_chain' not in st.session_state:
    # Initialize the question-answer chain and store its session state
    st.session_state['qa_chain'] = initialize_chain()
    st.session_state['messages'] = []

for message in st.session_state['messages']:
    # if the message is from  user display user's chat message
    if message['role'] == 'user':
        with st.chat_message("user"):
            st.markdown(message['content'])
    # if the message is from assistent display assistent's chat message
    else:
        with st.chat_message("assistant"):
            st.markdown(message['content'])

user_input = st.chat_input("Enter your question")

# If the user provides input add it to session state messages
if user_input:
    st.session_state['messages'].append({'role': 'user', 'content': user_input})

    # Display user's input in the chat
    with st.chat_message("user"):
        st.markdown(user_input)

    # Retrieve the QA chain from session state
    qa_chain = st.session_state['qa_chain']

    # Generate assistant's response
    with st.chat_message("assistant"):
        with st.spinner("Please wait while preparing answer..."):
            # Invoke the QA chain with user's question
            response = qa_chain.invoke({"question": user_input})
            # Display  assistant's response
            st.markdown(response["answer"])

    # Add assistant's response to session state
    st.session_state['messages'].append({'role': 'assistant', 'content': response["answer"]})
```

Init session state of the question-and-answer chain and a list of messages to qa_chain and a list for messages to maintain state across user interactions.

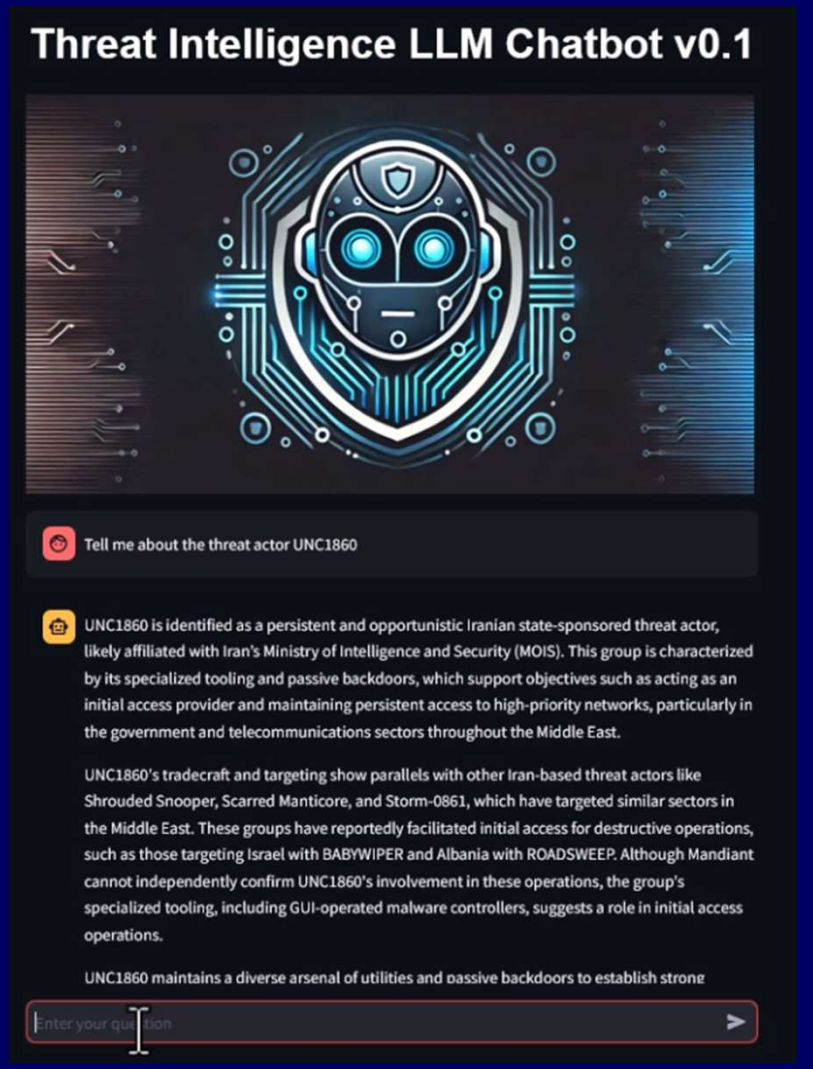
Messages are rendered with the roles user and assistant to differentiate their display styles.

The input box takes the user’s question for processing.

The assistant generates a response.

Questions and responses are stored in session state for persistence.

The CTI-Chatbot in action



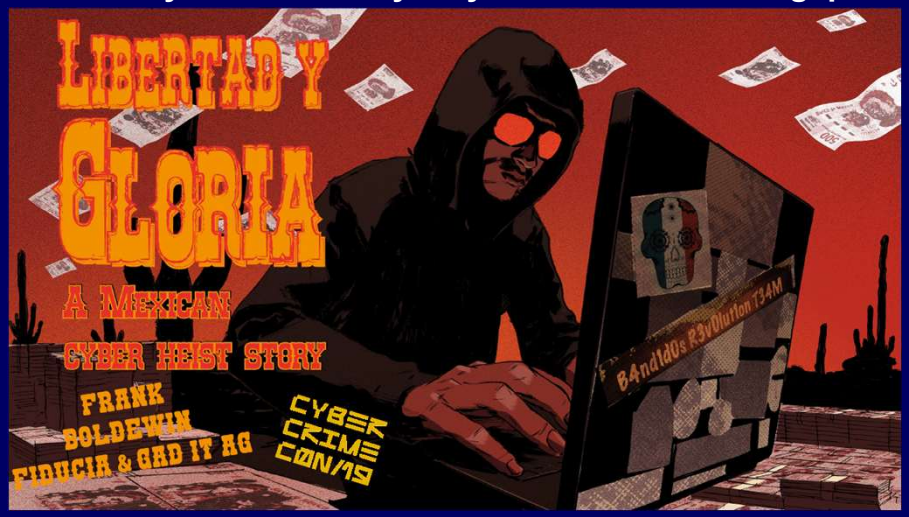
Video-Link: https://github.com/fboldewin/Empowering-CTI-with-AI-in-practice/raw/refs/heads/main/CTI Chatbot.mp4

Use Case 3: PDF content summarizer based on local LLM

Ollama

MISTRAL AI_

Source: https://github.com/fboldewin/Libertad-y-gloria---A-Mexican-cyber-heist-story---CyberCrimeCon19-Singapore/



The third use case shows how an LLM-based service can be implemented without relying on a cloud-based language model.

For this purpose, the Ollama platform is used, which enables the execution of AI models on local devices.

As LLM we use the freely available and powerful Mistral model (7,2 billion params.)

A PDF is used as the data source.
The content is a cybercrime case where the ATM infrastructure of a South American bank was attacked.

Setting up Ollama and other prerequesites

- Download+Install Ollama: https://ollama.com/download
- Pip Install → **langchain-ollama ollama**

- **Get the Mistral model**

```
C:\>ollama pull mistral:latest
pulling manifest
pulling ff82381e2bea...  11% ▕▕        ▕▕ 464 MB/4.1 GB  9.1 MB/s   6m41s
```

Developing a PDF content summarizer 1/2

```python
from langchain_community.document_loaders import PyPDFLoader
from langchain_ollama import OllamaEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from langchain_community.vectorstores import FAISS
import ollama

# Source: https://github.com/fboldewin/Libertad-y-gloria---A-Mexican-cyber-heist-story---CyberCrimeCon19-Singapore/
pdf = "MexicanATMHeist.pdf"

loader = PyPDFLoader(pdf)
pdf_text = loader.load()

tsplit = RecursiveCharacterTextSplitter(chunk_size = 500, chunk_overlap=50,  separators=['\n\n', '\n', ' ', ''])
raw_data = "".join(data.page_content for data in pdf_text)
chunks = tsplit.split_text(raw_data)

chunk_docs = []
for i, chunk in enumerate(chunks):
      chunk_docs.append(Document(page_content=chunk, metadata={'chunk': i}))

embeddings = OllamaEmbeddings(model="mistral:latest")

vectorstore = FAISS.from_documents(chunk_docs, embeddings)

retriever = vectorstore.as_retriever()
question = "Write a comprehensive report about the cyberattack."
context_object = retriever.invoke(question)
context = "".join(data.page_content for data in context_object)
```

Import needed modules

Use the Langchain PyPDFLoader to extract all texts from the PDF (For more advanced PDF parsers checkout PDFPlumber or Docling)

Chunk data and create document objects

Use mistral:latest model for embeddings

Create a FAISS vector database

Setup retriever and invoke

Developing a PDF content summarizer 2/2

```
prompt = f"""
You are a highly skilled Cyber Threat Intelligence Analyst specialized in identifying,
analyzing, and reporting on cyber threats.
Your task is to extract comprehensive, detailed, and actionable threat intelligence
based solely on the information provided in the context below.
Avoid assumptions or reliance on external sources,
base your analysis strictly on the provided information.

Context:
{context}

Your analysis must be clear, concise, and professional, use valid markdown formatting.

Question: {question}
"""

response = ollama.chat(model='mistral:latest',
                       messages=[{'role': 'user', 'content': prompt}],
                       options={"top_k": 10, "temperature": 0} )

with open("report.txt", "w", encoding="utf-8") as f:
    f.write(response['message']['content'])
```

Define prompt

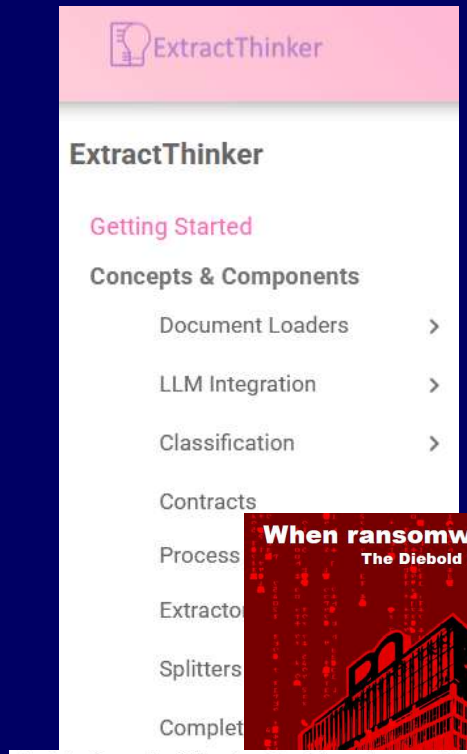
Use Ollama's chat API function with the mistral:latest LLM, configure top_k and temperature values

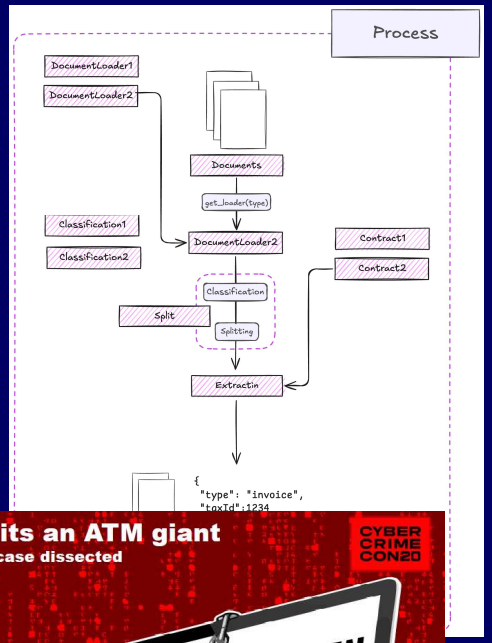
Write the generated report into a file

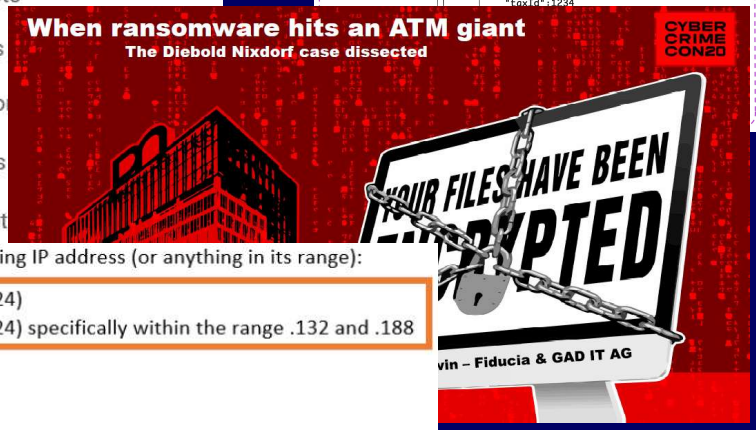
Developing a PDF content summarizer - Report output

```
Title: Comprehensive Report on Recent Cyberattack Targeting Mexican ATMs

  Introduction
  ----------
  A series of incidents involving malfunctioning Automated Teller Machines (ATMs) at two BBVA Bancomer branches in Mexico, which occurred on the
  4th and 5th of March, are indicative of a successful cyberattack. The analysis of available information suggests that this attack was carried out
  through malicious code injection via the Java Attach-API.

  Threat Overview
  ---------------
  The malware, once injected into the running Java Virtual Machine (JVM), gained access to all classes and methods within the self-service
  application. This indicates that the attack was targeted at a proprietary ATM application, not utilizing the usual XFS/JXFS API.

  Technique Analysis
  ------------------
  The malware's functionality includes calls to functions such as `getCashUnit`, `dispense`, `present`, and `waitForBillsTaken`. These function
  calls suggest that the attackers had control over the ATM's cash dispensing mechanism. Additionally, after dispensing cash, a status report was
  sent to an attacker-controlled server.

  Impact Analysis
  ---------------
  The consequences of this cyberattack were significant, as it resulted in the unauthorized dispensation of cash from ATMs. While the exact amount
  stolen is not specified, it is mentioned that luxury car purchases and remittances of millions were involved.

  Investigation Update
  --------------------
  Following the incidents, the police initiated an investigation, employing covert surveillance and remote observation of several suspects. They
  gathered enough evidence to execute search and arrest warrants in Leon. It is also rumored that the police received clues from an anonymous source.

  Mitigation Recommendations
  --------------------------
  To prevent similar incidents, it is recommended to:

  - Enhance security awareness training within the organization, emphasizing the risks associated with unintentionally uploading company-related
  files before third parties find them.
  - Proactively hunt for such files on VirusTotal (VT) and take necessary actions to remove them if found.
  - Implement strict access controls and regular audits of ATM applications, ensuring they are not vulnerable to similar attacks.
  - Monitor network traffic for any unusual activity related to ATMs, particularly status reports after cash dispensation.

  Conclusion
  ----------
  The recent cyberattack on Mexican ATMs serves as a stark reminder of the ever-evolving threat landscape in the digital world. Organizations must
  remain vigilant and proactive in their approach to cybersecurity, focusing on threat intelligence, incident response, and recovery strategies.
```

Use Case 4: Extracting IoCs from threat reports







Activity from the following IP address (or anything in its range):

- 172.2.231.27 (..../24)
- 172.241.27.0 (..../24) specifically within the range .132 and .188

QakBot Payload Site:
sollight.com[.]hk

The fourth use case utilizes the Intelligent Document Processing (IDP) framework Extract Thinker in order extract Indicators of Compromise (IoCs) from a threat report.
https://enoch3712.github.io/ExtractThinker

The framework comes with a set of document loaders and classification techniques, LLM integration and workflow processing.

As a showcase, an analysis of a ransomware incident at the ATM manufacturer Diebold-Nixdorf serves as an example, which contains IoCs both in the texts and images.

Use Case 4: Extracting IoCs from threat reports

```python
import warnings
warnings.filterwarnings("ignore")

import re, os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "gemini-key.json"

from docling.datamodel.pipeline_options import (
    PdfPipelineOptions,
    TesseractCliOcrOptions,
    TableStructureOptions,
)

from enum import Enum
from typing import List, Optional
from pydantic import BaseModel, Field
from extract_thinker import Contract, Extractor

from docling.datamodel.base_models import InputFormat
from docling.document_converter import PdfFormatOption

from extract_thinker.document_loader.document_loader_docling import DocumentLoaderDocling, DoclingConfig

from extract_thinker.models.completion_strategy import CompletionStrategy

class Types(str, Enum):
    EMAIL = "email"
    DOMAIN = "domain"
    IP = "ip"
    URL = "url"
    HASH = "hash"
    FILE = "file"

class Indicator(BaseModel):
 IOCType: Optional[str] = Field(None, description="Indicator of Compromise type like email, domain, ip, url, hash, file")
 IOCValue: Optional[str] = Field(None, description="IOC Value like 'APT29.com', 'evil@apt29.ru', '31.3.3.7', etc.")
 IOCContext: Optional[str] = Field(None, description="Context information to IOC")

class IndicatorsContract(BaseModel):
    Indicators: Optional[List[Indicator]] = Field(None, description="List of all IOCs")
```

Next to ExtractThinker modules, we also import the powerful Docling library for document processing.

Define six types of IoCs to be extracted from the threat report.

Define an indicator class based on the Pydantic BaseModel (named Contracts within ExtractThinker) with three fields, IoC Type, IoC value and a IoC context information and a class representing a collection of IoCs, containing a field to a list of indicator objects.

Use Case 4: Extracting IoCs from threat reports

```
prompt = """You are an experienced Cyber Threat Intelligence and Cyber Defense Analyst.
Get all available Indicators of Compromise (IoCs) and provide a precise description about every IoC. You MUST not skip an IoC.
The following IoC types should be considered: """ + ", ".join([ioc_type.value for ioc_type in Types])

tesseract_path = os.getenv("TESSERACT_PATH")

ocr_options = TesseractCliOcrOptions(force_full_page_ocr=True, tesseract_cmd=tesseract_path)

table_options = TableStructureOptions(do_cell_matching=True)

pdf_options = PdfPipelineOptions(
    do_table_structure=True,
    do_ocr=True,
    ocr_options=ocr_options,
    table_structure_options=table_options
)

format_options = {
    InputFormat.PDF: PdfFormatOption(pipeline_options=pdf_options)
}

config = DoclingConfig(format_options=format_options,ocr_enabled=True,force_full_page_ocr=True)

loader = DocumentLoaderDocling(config)

loader.set_vision_mode(True)

CTIRPT = "DN-Ransomware-Case.pdf"

content = loader.load(CTIRPT)

extractor = Extractor()
extractor.load_document_loader(loader)
extractor.load_llm("vertex_ai/gemini-2.5-flash-preview-04-17")

xtracted_iocs = extractor.extract(
    CTIRPT,
    IndicatorsContract,
    vision=True,
    content=prompt,
    completion_strategy=CompletionStrategy.PAGINATE
)
```

Define the llm prompt by role, precise instructions and retrieval of the IoC values by iterating through the type enums.

Configure Docling format options and image visioning.

Init the Extractor component and use gemini-2.5 flash preview as multimodal model.

Extract IoCs and use the PAGINATE completion strategy to process multi-page documents in parallel.

Use Case 4: Extracting IoCs from threat reports

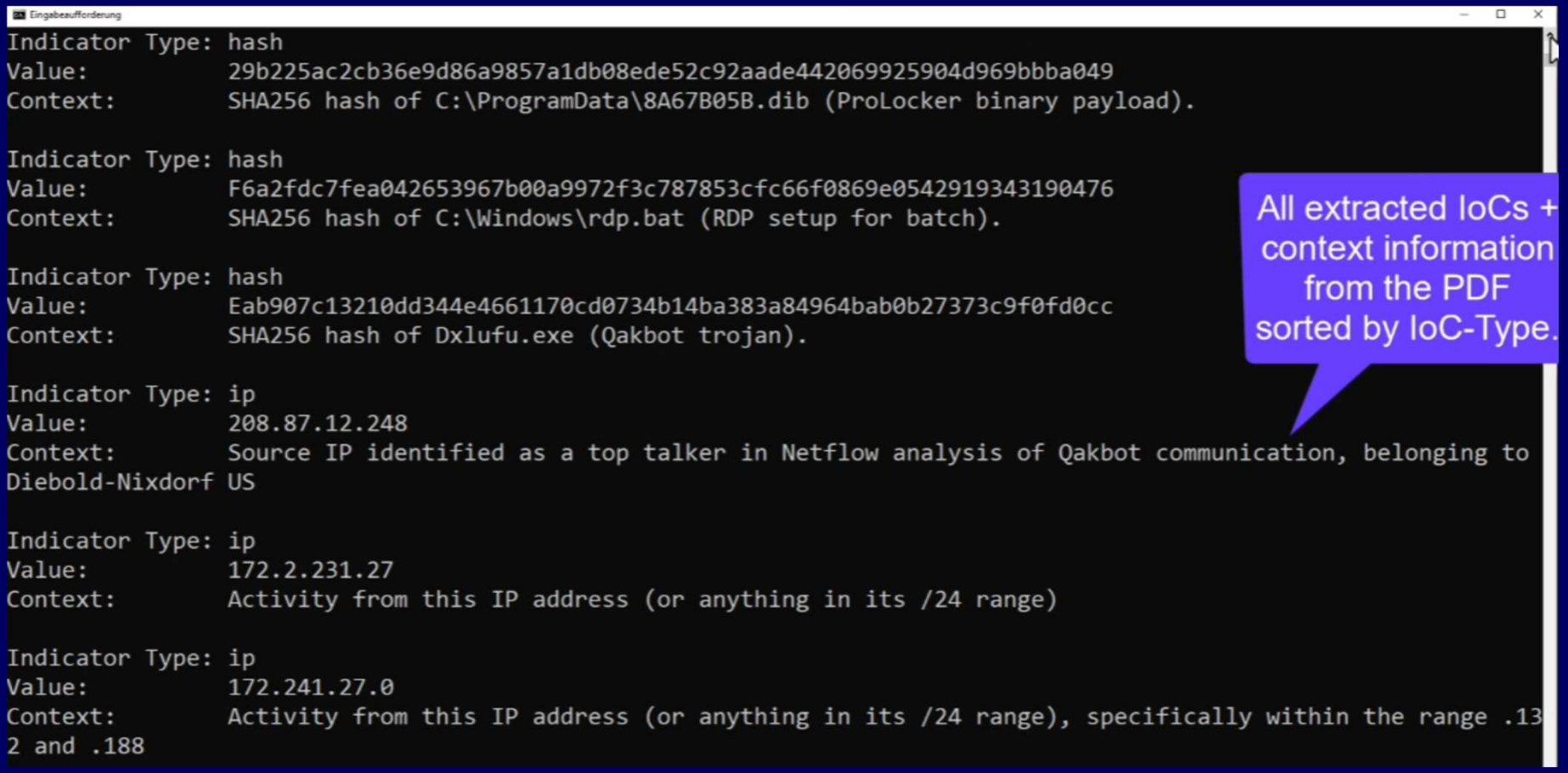
```
ioc_validation_patterns = {
    "email": re.compile(
        r'^[^@\s]+@[^@\s]+\.[^@\s]+$'),
    "ip": re.compile(
        r'^(25[0-5]|2[0-4]\d|[01]?\d?\d)\.'
        r'(25[0-5]|2[0-4]\d|[01]?\d?\d)\.'
        r'(25[0-5]|2[0-4]\d|[01]?\d?\d)\.'
        r'(25[0-5]|2[0-4]\d|[01]?\d?\d)(?:/\d{1,2})?$'),
    "domain": re.compile(
        r'\b(?:(?:[a-zA-Z0-9-]+\.)+(?:[a-zA-Z]{2,})|[a-z2-7]{16}\.onion|[a-z2-7]{56}\.onion)\b'),
    "url": re.compile(
    r'^(?:https?://(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}(?::\d+)?(?:/[^\s]*)?'
    r'|https?://[a-z2-7]{16}\.onion(?::\d+)?(?:/[^\s]*)?'
    r'|https?://[a-z2-7]{56}\.onion(?::\d+)?(?:/[^\s]*)?'
    r'|[a-z2-7]{16}\.onion'
    r'|[a-z2-7]{56}\.onion)$'),
    "hash": re.compile(
        r'^([a-fA-F0-9]{32}|[a-fA-F0-9]{40}|[a-fA-F0-9]{64})$'),
    "file":re.compile(
        r'(?:\*|(?:\/|(?:[a-zA-Z]:\\))?(?:[a-zA-Z0-9_\-\. ]+[\/\\])*[a-zA-Z0-9_\-\. ]+\.[a-zA-Z0-9]{1,10})')
}

seen = set()
unique_indicators = []

for indicator in xtracted_iocs.Indicators:
    ioc_type = indicator.IOCType
    ioc_value = indicator.IOCValue
    if ioc_type in ioc_validation_patterns:
        pattern = ioc_validation_patterns[ioc_type]
        if pattern.fullmatch(ioc_value):
            key = (ioc_type, ioc_value)
            if key not in seen:
                seen.add(key)
                unique_indicators.append(indicator)

unique_indicators.sort(key=lambda x: x.IOCType)

for ioc in unique_indicators:
    print(f"""\
Indicator Type:\t{ioc.IOCType}
Value:         \t{ioc.IOCValue}
Context:       \t{ioc.IOCContext}
""")
```

Validate IoCs by its type with regex.

Remove potential duplicates.

Print extracted IoCs and its context information.

Use Case 4: The IoC extraction agent in action



Video-Link: https://github.com/fboldewin/Empowering-CTI-with-AI-in-practice/raw/refs/heads/main/PDF-IoC-Extract.mp4

Key takeaways

1. **AI** and **LLMs** can make a significant **contribution** by **supporting** and **quality-enhancing** the **work** of a threat intelligence **analyst**.

2. **Careful data preparation** and **encoding** in your vector database is the **key to** any **successful RAG application**. **Equally important** is selecting the right **model** for your use case, **tuning parameters** such as temperature and top K and developing precise **prompts** to steer the system towards the desired results.

3. **Ensure** to **secure** your **LLM apps** (another talk) to **protect against** a variety of **attack vectors**
→ https://genai.owasp.org/llm-top-10/

4. These **advantages** can also be **easily adapted** to **other security areas**, such as **incident response** and **forensic analyses**, **anomaly detection** or to **assist** in **red teamings**.

Where human expertise and AI converge - securing the future together