✅ Ultimate Goal

Build a Threat Intelligence Processing Pipeline:

- That can automatically ingest cyber threat data (IPs and URLs)
- Normalize, enrich, filter, and store that data
- Expose the cleaned data through a RESTful API built in Flask

This is similar to how real-world cybersecurity systems (like SOCs or SIEMs) work.

📁 Main Components (5 Parts)

1️⃣ Threat Feed Ingestion Module (Python)

- Python script that downloads and reads data from 3 public feeds:
  - http://www.blocklist.de/lists/apache.txt — List of malicious IPs
  - http://www.spamhaus.org/drop/drop.txt — IPs or IP subnets
  - https://osint.digitalside.it/Threat-Intel/lists/latesturls.txt — Malicious URLs

What “normalize” means:

- All data should be converted into a standard format, regardless of the source.

Example Schema:

json

```
{
  "type": "ip" or "url",
  "value": "1.2.3.4" or "http://bad.example.com/drop.exe",
  "source": "spamhaus", "blocklist", or "digitalside",
  "timestamp": "2025-07-31T18:30:00Z"
}
```

Why this matters:

- Different sources provide different formats. Normalizing ensures everything looks consistent so your pipeline can process it easily.

2️⃣ Enrichment & Filtering Module (Python)

🔍 IP Enrichment:

- Use ipinfo.io or a local mock database to get metadata:
  - Country
  - City
  - ISP

- Organization

Why mock?

- Public APIs like ipinfo.io have request limits or require API keys.
- A local mock_ip_db.json file with a few sample entries is acceptable.

🔗 URL Filtering:

- Focus only on suspicious file types (like malware):
  - .exe, .zip, .scr, etc.
- we can do this using string checking in Python:
  if url.endswith('.exe')

🧠 Bonus: Mini ML Classifier (Optional)

- we can write a small classifier to label URLs as:
  - suspicious
  - benign
- Example approaches:
  - Rule-based (e.g., URL has many numbers, weird domains)
  - Logistic regression based on simple features (length, keywords like download, free, crack, etc.)

🚫 De-duplication:

- Remove repeated indicators of compromise (IOCs).
- You can use sets or hash comparisons.

3️⃣ Storage Layer

Choose one method to store your processed data:

Options:

- JSON file (easiest to implement and test)
- MongoDB (NoSQL database, good for JSON-like data)
- Elasticsearch (search-optimized DB)

For simplicity, use a JSON file like threat_data.json:

json

```
[
  {
```

"type": "ip",
"value": "1.2.3.4",
"source": "spamhaus",
"location": "India",
"isp": "Airtel"
},
...
]

4️⃣ API Interface (Python Flask)

create a simple Flask REST API to expose the stored IOCs:

Required Endpoints:

Method	Endpoint	Function
GET	/iocs	Returns all indicators (IPs + URLs)
GET	/iocs?type=ip	Filters only IPs
GET	/iocs?source=spamhaus	Filters by source
POST	/refresh	Triggers the ingestion & enrichment pipeline again

The API does not need a database. You can read from the saved JSON file instead.

5️⃣ Architecture Documentation (README or ARCHITECTURE.md)

- Module Overview: Describe each part of your code
- Flow Diagram: Simple visual or bullet explanation of:
    - Feed fetch → Normalization → Enrichment → Store → API expose
- How to Scale:
    - Use cron jobs to fetch regularly
    - Add a message queue (like RabbitMQ) for tasks
    - Add caching
    - Move to cloud (e.g., AWS Lambda or ECS)

Also explain:

- Any assumptions you made (e.g., IP enrichments mocked)
- Any challenges (e.g., rate limits, parsing issues)

⏱️ Time Expectations

- Suggested time: 6–8 hours
- Deadline: Up to 4 days

🧪 Evaluation Criteria

They will look at:

✅ Clean and modular Python + Flask code
✅ Handling edge cases: bad data, timeouts
✅ Good folder structure (like ingestion/, api/, data/, etc.)
✅ Well-written README with good explanations
✅ Bonus ML classifier or smart filtering
✅ Clear naming conventions and logging

✅ Step 1: Threat Feed Ingestion and Normalization

🎯 Goal

- Download the 3 feeds from public URLs
- Parse them
- Normalize all entries into a common JSON schema

Feed Name	URL	Type
Blocklist.de	http://www.blocklist.de/lists/apache.txt	List of individual IPs
Spamhaus	http://www.spamhaus.org/drop/drop.txt	IPs or subnets
DigitalSide OSINT	https://osint.digitalside.it/Threat-Intel/lists/latesturls.txt	Malicious URLs

File name: Ingestion/fetch-blocklist.py

This Python file is part of a **threat intelligence ingestion pipeline**. Its main purpose is to:

🔍 **Fetch and parse a list of suspicious IP addresses** from blocklist.de, specifically from their apache.txt feed, which typically contains IPs involved in brute force attacks on Apache servers.

What the code does (step-by-step):

1. Defines the feed URL:

BLOCKLIST_URL = "http://www.blocklist.de/lists/apache.txt"

This is a publicly available list of IPs flagged for brute-force attempts against Apache web servers.

---

2. Main function: fetch_blocklist_feed()

- Downloads the list from the URL.
- Parses it line by line.
- Validates each line to check if it's a properly formatted IP address.
- Returns a list of IOC (Indicator of Compromise) dictionaries like:

```
{
  'value': '1.2.3.4',
  'type': 'ip',
```

'source': 'blocklist',
'source_url': BLOCKLIST_URL,
'category': 'brute_force',
'raw_data': '1.2.3.4',
'line_number': 17
}

---

3. Helper function: is_valid_ip(ip)

- Uses regex to check if a string looks like an IPv4 address.
- Further validates that each number (octet) is between 0 and 255.

📌 Use Cases:

- Used in **cybersecurity tools** or **SIEM pipelines** to enrich logs with threat intelligence.
- Helps **block malicious IPs** proactively on firewalls, IDS/IPS, etc.
- Can be integrated into a **threat feed ingestion system**.

🧠 Summary:

This file automates the **download, validation, and normalization** of IP addresses from a known threat feed and converts them into structured threat data (IOCs) for further use in a security system.

File name: Ingestion/fetch-digitalside.py

This Python file is part of a **threat intelligence ingestion system**, and it focuses on fetching and parsing **malicious URLs** from **DigitalSide's open-source threat feed**.

✅ Purpose:

To **automatically download, validate, and normalize** URLs known to be malicious (e.g., phishing, malware delivery) from this feed:

bash

CopyEdit

https://osint.digitalside.it/Threat-Intel/lists/latesturls.txt

---

What it does step by step:

1. Sets up logging and the threat feed URL:

DIGITALSIDE_URL = "https://osint.digitalside.it/Threat-Intel/lists/latesturls.txt"

This file is hosted by DigitalSide and regularly updated with malicious URLs.

---

2. Main function: fetch_digitalside_feed()

- Downloads the feed using requests.
- Reads the text content line by line.
- Skips empty lines and comments (those starting with #).
- Validates each line to ensure it's a proper URL using is_valid_url().
- Parses the URL to extract components like:
  - scheme (e.g., http, https)
  - domain (e.g., malicious.com)
  - path (e.g., /virus.exe)

Each valid URL is converted into a structured **IOC (Indicator of Compromise)** dictionary like this:

```
{
  'value': 'http://malicious-site.com/bad.exe',
  'type': 'url',
  'source': 'digitalside',
  'source_url': 'https://osint.digitalside.it/Threat-Intel/lists/latesturls.txt',
  'category': 'malware',
  'raw_data': 'http://malicious-site.com/bad.exe',
  'line_number': 42,
  'domain': 'malicious-site.com',
  'path': '/bad.exe',
  'scheme': 'http'
}
```

---

3. Helper function: is_valid_url(url)

Uses Python’s urlparse() to check if a string:

- Has a valid **scheme** (http, https, etc.)
- Has a **network location** (i.e., a domain name or IP)

🧠 **Why these matters (Use Cases):**

- This feed is useful in **cybersecurity tools** to detect and block access to known malicious domains/URLs.
- The parsed IOCs can be fed into:
  - SIEM systems
  - Threat intel platforms
  - Firewalls / DNS filters
  - Malware detection engines

🧪 **Summary:**

This script fetches a list of **malicious URLs** from DigitalSide's threat feed, validates and parses each URL, and converts them into structured threat intelligence data that can be used by cybersecurity systems.

File name: Ingestion/fetch-spamhaus.py

This Python file is designed to **fetch, parse, and normalize threat intelligence data** from the **Spamhaus DROP list**, which contains **IP address ranges (in CIDR notation)** associated with **malicious or botnet-related activity**.

✅ **Purpose:**

To convert Spamhaus's **DROP (Don't Route Or Peer) list** into structured IOCs (Indicators of Compromise) that can be used by security systems for:

- Blocking known bad IP ranges
- Enriching threat intelligence databases
- Feeding into firewalls or SIEM platforms

🔧 **What it does (step-by-step):**

1. Feed source defined:

python

CopyEdit

SPAMHAUS_URL = "http://www.spamhaus.org/drop/drop.txt"

This is the official feed of CIDR blocks (IP ranges) that are dangerous or controlled by bad actors.

---

2. Main function: fetch_spamhaus_feed()

- Fetches the file via HTTP.
- Parses the content line by line.
- Skips:
  - Empty lines
  - Comments (lines starting with ;)
- For each valid line:
  - Extracts the **CIDR block** (e.g., 123.45.67.0/24)
  - Optionally extracts the **SBL reference** (Spamhaus Block List ID)
- Validates the CIDR format using the is_valid_cidr() function.
- Converts each valid entry into a structured dictionary like:

python

CopyEdit

```
{
  'value': '123.45.67.0/24',
  'type': 'ip',
  'source': 'spamhaus',
  'source_url': 'http://www.spamhaus.org/drop/drop.txt',
  'category': 'botnet_range',
  'raw_data': '123.45.67.0/24 ; SBL123456',
  'line_number': 10,
  'sbl_reference': 'SBL123456'
}
```

---

3. Helper function: is_valid_cidr(cidr)

- Uses a regex pattern to match a CIDR (e.g., 192.168.0.0/24)
- Checks:
  - All octets are between 0–255
  - Prefix is between 0–32

📌 **Use Cases:**

- Import these IOCs into:

- Firewalls to block traffic from these IP ranges
  - IDS/IPS for alerting
  - SIEMs for enrichment or detection
- Helps in protecting infrastructure from **botnets, spam, malware**, and other malicious activities.

🧠 Summary:

This file downloads and parses the **Spamhaus DROP list**, validates each IP range (CIDR), and structures the data into IOCs that identify **malicious IP ranges linked to botnets or spam operations**. It can be used in threat detection and prevention pipelines.

File name: Ingestion/normalize.py

This Python file is part of a **threat intelligence processing pipeline**, and it is responsible for **normalizing** IOCs (Indicators of Compromise) — such as malicious IPs, URLs, or CIDR blocks — into a **standardized format** that can be used consistently across a security system.

✅ Purpose:

To take in **raw threat data** (from sources like Spamhaus, DigitalSide, or blocklist.de) and convert each entry into a **clean, uniform, enriched, and structured format**.

🔧 What it does (Step-by-step):

1. Function: normalize_iocs(raw_iocs)

- Input: a list of raw IOC dictionaries from different sources.
- Output: a list of **normalized IOCs**, where each IOC has the same structure.
- Adds useful metadata and confidence scores to each IOC.
- Includes error handling and logging.

Each **normalized IOC** looks like:

python

CopyEdit

```
{
  'id': 'unique_id_hash',
  'value': 'http://malicious.com/evil.exe',
  'type': 'url',
```

    'source': 'digitalside',
    'source_url': 'https://osint.digitalside.it/...',
    'category': 'malware',
    'first_seen': '2025-08-01T07:12:00.000Z',
    'last_updated': '2025-08-01T07:12:00.000Z',
    'confidence': 0.8,
    'metadata': {
        'raw_data': 'http://malicious.com/evil.exe',
        'line_number': 14,
        'domain': 'malicious.com',
        'path': '/evil.exe',
        'scheme': 'http'
    }
}

🧩 Key Helper Functions:

generate_ioc_id(ioc)

- Generates a unique ID (MD5 hash) based on the IOC's value, type, and source.

calculate_confidence(ioc)

- Assigns a **confidence score (0.0 – 1.0)** based on:
    - The source (e.g., Spamhaus = 0.9)
    - The type (e.g., CIDR = more confidence)
    - Suspicious traits (e.g., .exe or .zip in URLs)

extract_metadata(ioc)

- Extracts and organizes **extra information**:
    - Line number
    - Raw feed data
    - URL components (domain, path, scheme)
    - SBL reference for Spamhaus

🧠 Why it's useful:

- Makes IOCs easier to store in a database or pass into downstream security tools.
- Enables consistent processing regardless of the data source.
- Adds valuable enrichment like confidence scores and parsed URL components.

📌 **Summary:**

This file **standardizes and enriches raw IOCs** from multiple sources into a unified schema with consistent fields, unique IDs, confidence scores, and useful metadata. It's essential for any system that aggregates or processes threat intelligence feeds.

🚀 Step 2: Enrichment & Filtering

Module	Purpose
enrichment/enrich_ip.py	Add info like ISP, country, threat labels
enrichment/filter_urls.py	Remove clean URLs, keep only suspicious ones
enrichment/deduplicate.py	Remove exact duplicates
enrichment/mlclassifier.py	Heuristic Approach

File name: enrichment/dedublicate.py

This Python file is a **deduplication module for IOCs (Indicators of Compromise)**. It's part of a **threat intelligence pipeline** and is responsible for:

✅ Main Purpose

Removing **duplicate IOCs** and optionally **merging their metadata** to retain important information — like sources, confidence, and enrichment — in a clean, single version of each unique IOC.

📦 What Are IOCs?

IOCs are pieces of threat data such as:

- IP addresses
- URLs
- Domains

- File hashes
  They help identify malicious activity.

🔍 What the File Contains

1. deduplicate_iocs()

Basic deduplication function.

- **Input**: A list of IOCs (dictionaries).
- **How it works**: Builds a key from value:type, and checks for repeats.
- **If duplicate**: Calls merge_duplicate_metadata() to merge data.
- **Logs how many were removed**.

---

2. merge_duplicate_metadata()

Merges data from a duplicate IOC into the first one seen:

- Combines sources
- Keeps the higher confidence score
- Merges any enrichment fields (e.g., location, ASN, malware family)
- Adds or increments a duplicate_count field

This ensures **no data is lost**, even when duplicates are removed.

---

3. deduplicate_by_normalized_value()

More advanced deduplication based on **normalized** IOC values.

- Uses normalize_ioc_value() to standardize each IOC before comparison.
- Example: http://malicious.com/ and http://malicious.com → considered same

This helps detect **logical duplicates** that aren't **textually identical**.

---

4. normalize_ioc_value()

Normalizes IOC values:

- For **URLs**:
  - Lowercases
  - Strips trailing slashes /
  - Removes query params like ?utm=abc
- For **IPs**:

- Strips extra spaces

This ensures different formatting doesn't trick the system into thinking they're different IOCs.

🧠 Example

Input:

python

CopyEdit

```
[
 {"value": "http://bad.com/", "type": "url", "source": "blocklist"},
 {"value": "http://bad.com", "type": "url", "source": "spamhaus", "confidence": 0.9}
]
```

Output:

python

CopyEdit

```
[
 {
  "value": "http://bad.com/",
  "type": "url",
  "source": "blocklist",
  "sources": ["blocklist", "spamhaus"],
  "confidence": 0.9,
  "duplicate_count": 2
 }
]
```

✅ Summary

Function	Purpose
deduplicate_iocs()	Basic deduplication by value:type
merge_duplicate_metadata()	Merges data from duplicate IOCs
deduplicate_by_normalized_value()	Dedupes using normalized values

Function	Purpose
normalize_ioc_value()	Standardizes IOC values for comparison

This module is critical for ensuring your **threat data isn't bloated or misleading** by duplicates — while still preserving valuable metadata.

File name: enrichment/enrich_ip.py

This Python file is responsible for **enriching IP address IOCs** (Indicators of Compromise) with **additional context** such as:

- 🌍 Geolocation (country, city, region)
- 🏢 ISP or organization
- ⏱️ Timezone
- 🌐 Public/private/special classification

✅ Purpose of the File

Add context to IP addresses in your threat intel pipeline to make them more useful for **analysis**, **correlation**, or **decision-making**.

🔍 Key Functions & What They Do

1. enrich_ip_iocs(iocs: list)

- Main function that takes a list of IOCs.
- For every IOC of type ip, it:
    - Tries to extract and enrich the IP address.
    - Adds enrichment data to the IOC as a new field: ioc['enrichment'].

2. enrich_single_ip(ip_address, mock_db)

- Enriches a **single IP address** by:
    1. Checking if enrichment exists in a **mock local database**.
    2. If not, and an IPINFO_API_KEY is available, it uses **ipinfo.io**.
    3. If that fails, falls back to **basic classification** (e.g., is the IP public or private).

3. load_mock_ip_db()

- Loads a local JSON file: data/mock_ip_db.json.
- This mock database contains enrichment data like:

json

CopyEdit

```json
{
  "8.8.8.8": {
    "country": "US",
    "org": "Google LLC",
    ...
  }
}
```

4. get_mock_enrichment(ip_address, mock_db)

- Looks up a given IP in the loaded mock DB and returns enrichment data if available.

5. get_ipinfo_enrichment(ip_address, api_key)

- Uses the **ipinfo.io API** (if API key is available) to fetch:
  - country, region, city, org, timezone, etc.
- Adds source: "ipinfo.io" to indicate where enrichment came from.

6. get_basic_enrichment(ip_address)

- If all else fails, uses basic logic to classify the IP as:
  - private (e.g., 10.0.0.0/8, 192.168.0.0/16)
  - special (e.g., 127.0.0.1, 169.254.0.0/16)
  - public
- Adds source: "basic_analysis" to mark the fallback logic.

🧠 Why Is This Useful?

IP enrichment is critical in threat intelligence because:

- Knowing an IP is **from Russia** or belongs to **a known hosting provider** helps assess risk.
- Private or special IPs are likely **internal**, not external threats.
- Helps analysts **prioritize** or **filter** IOCs more intelligently.

📦 Example Input

python

CopyEdit

```
[
 {"type": "ip", "value": "8.8.8.8", "source": "spamhaus"}
]
```

🧾 **Example Output After Enrichment**

python

CopyEdit

```
[
 {
  "type": "ip",
  "value": "8.8.8.8",
  "source": "spamhaus",
  "enrichment": {
   "country": "US",
   "region": "California",
   "city": "Mountain View",
   "org": "Google LLC",
   "timezone": "America/Los_Angeles",
   "source": "ipinfo.io"
  }
 }
]
```

✅ **Summary**

Function	Purpose
enrich_ip_iocs()	Main enrichment loop for all IP IOCs
enrich_single_ip()	Tries mock DB → ipinfo.io → basic fallback
load_mock_ip_db()	Loads local mock enrichment data
get_ipinfo_enrichment()	Calls external API for detailed info
get_basic_enrichment()	Last-resort logic using IP address structure

⚡ Time Comparison

Method	Avg Time per Lookup	Notes
Local DB	**1–5 ms**	Fastest, especially if in RAM
ipinfo.io API	**100–400 ms**	Slower due to network + HTTPS

File name: enrichment/filter_urls.py

This Python file is a **URL filtering and enrichment module** for a threat intelligence pipeline. It analyzes **URL-type IOCs** (Indicators of Compromise) and decides whether they are **suspicious or not**, using heuristics like file extensions, patterns, domains, and structure.

✅ Purpose

To identify and **filter out benign URLs**, and retain only those that are potentially **malicious**, **suspicious**, or **high-risk**.

🔍 Key Functionalities

1. filter_suspicious_urls(iocs: List) -> List

Main function that:

- Filters and enriches **IOC entries with type == 'url'**
- Calculates a **suspicion score**
- Marks each URL as:
    - is_suspicious: True or False
    - suspicion_score: 0.0 – 1.0
    - suspicious_indicators: a list of flags (e.g. "suspicious_extension:.exe")
- Keeps or discards URLs based on heuristics

---

2. calculate_url_suspicion(url: str) -> float

Calculates a **numeric suspicion score** based on:

Feature	Score Added
Suspicious file extension (e.g. .exe)	+0.3
Suspicious domain or keyword pattern	+0.2 (each)
IP address as domain	+0.4

Feature	Score Added
Long domain or long URL	+0.1–0.2
Suspicious ports (8080, 8443, 9999)	+0.1
Suspicious paths (e.g., /admin)	+0.1
Fails URL parsing	+0.3

Final score is capped at 1.0.

---

3. get_suspicious_indicators(url: str) -> List[str]

Extracts **human-readable tags** that explain **why** the URL is suspicious.

Example:

json

CopyEdit

```json
[
  "suspicious_extension:.exe",
  "pattern:suspicious_keywords",
  "long_url",
  "suspicious_port:8080"
]
```

---

4. should_keep_url(url: str, suspicion_score: float) -> bool

Determines whether the suspicious URL should be **kept for further processing**:

- ✅ Keep if:
  - Contains **high-risk extensions** (e.g., .exe, .zip, .bat, .scr)
  - Has **suspicion score > 0.4**
  - Contains an **IP address** in the domain
- ❌ Otherwise: discard

🧠 Why This Is Useful

This module helps:

- 🔍 **Reduce noise** in large IOC datasets by eliminating obviously harmless URLs
- 🎯 Focus analyst or automation effort on the **most dangerous or suspicious links**
- 🧱 Enrich threat intel by tagging IOCs with **metadata about malicious indicators**

📦 Example Input

python

CopyEdit

```
[
 {'type': 'url', 'value': 'http://example.com/download/file.exe'},
 {'type': 'url', 'value': 'https://google.com'},
 {'type': 'ip', 'value': '8.8.8.8'}
]
```

📤 Example Output

python

CopyEdit

```
[
 {
  'type': 'url',
  'value': 'http://example.com/download/file.exe',
  'enrichment': {
   'suspicion_score': 0.6,
   'is_suspicious': True,
   'suspicious_indicators': [
    'suspicious_extension:.exe',
    'pattern:suspicious_keywords'
   ]
  }
 },
 {
  'type': 'ip',
```

'value': '8.8.8.8'

}

]

✅ Summary Table

Function Name	**Role**
filter_suspicious_urls	Enrich and keep only suspicious URLs
calculate_url_suspicion	Compute a score based on structure and patterns
get_suspicious_indicators	List textual reasons why the URL is considered suspicious
should_keep_url	Decide whether to keep or drop the URL

File name: enrichment/ml_classifier.py

🔍 What It Does:

🧠 Purpose:

Trains a **very basic keyword-based model** from a list of URLs labeled as either **malicious** (True) or **benign** (False). It identifies which keywords are more likely to appear in malicious URLs vs benign ones.

📥 Inputs:

- training_urls: A list of tuples like:

python

CopyEdit

```
[
  ("http://malware-site.com/virus", True),
  ("https://docs.microsoft.com/support", False)
]
```

Each tuple is:

  - A URL (string)
  - A label (True if malicious, False if benign)

📤 Outputs:

Returns a dictionary of statistics including:

- Total URLs
- Count of malicious and benign URLs
- Top 20 most frequent **malicious** and **benign** keywords
- Top 50 **keywords ranked by maliciousness score**

🔧 How It Works:

1. Separates URLs into malicious and benign based on the label

2. Tokenizes the URLs into words using regex (\b\w+\b)

E.g., "http://malware.com/virus" → ["http", "malware", "com", "virus"]

3. Counts frequency of each keyword in:

- malicious URLs → malicious_keywords
- benign URLs → benign_keywords

4. Calculates a score for each keyword:

python

CopyEdit

```
score = malicious_frequency / benign_frequency
```

- If a word appears more in malicious URLs, it gets a **higher score**
- If it appears only in benign URLs → score is 0
- If it appears **only** in malicious URLs → score = mal_freq (since benign = 0)

5. Sorts the keywords by score and keeps the top 50.

📊 Example Output:

python

CopyEdit

```
{
  'total_urls': 100,
  'malicious_count': 30,
  'benign_count': 70,
  'top_malicious_keywords': [('virus', 15), ('malware', 12), ...],
  'top_benign_keywords': [('support', 20), ('docs', 18), ...],
  'keyword_scores': {
```

```
      'virus': 5.0,
      'malware': 4.0,
      'download': 3.2,
      ...
    }
}
```

✅ Use Case:

This function helps **discover keywords** you might want to include in your MALICIOUS_KEYWORDS or BENIGN_KEYWORDS dictionary used for scoring.

🚀 Step 3: Storage Layer:

🔄 Options:

1. JSON File Storage (Simple & Local)

- 🔸 Save IOCs to a .json file
- ✅ Good for local testing
- ❌ No real-time search or scale

---

2. MongoDB (Recommended for Demo/POC)

- 📦 Document-based NoSQL database
- 🔍 Supports filtering, searching, and updating IOCs
- 🧪 Great for frontend analytics + filtering

---

3. Elasticsearch (Advanced)

- ⚡ For full-text search + filtering
- 📈 Used in enterprise SOC dashboards
- 🧰 Requires more setup

File name: storage/save_data.py

✅ Purpose of the File

To **persist** the results of your pipeline into structured JSON files for:

- Storage
- Auditing

- Debugging
- Future re-use

🔍 Key Functions Explained

---

1. save_processed_iocs(iocs, metadata=None)

✅ **What it does:**

- Saves the **final list of enriched, deduplicated, filtered IOCs** to:
  - data/processed_iocs.json → main/latest file
  - data/processed_iocs_backup_<timestamp>.json → time-based backup

🧠 **Why:**

- Ensures the most recent result is accessible
- Maintains historical versions for rollback or comparison

---

2. save_raw_feed_data(feed_name, raw_data)

✅ **What it does:**

- Saves **raw text content** from feeds (e.g., blocklist, spamhaus) into:
  - data/raw/<feed_name>_<timestamp>.txt
  - data/raw/<feed_name>_latest.txt

🧠 **Why:**

- Useful for **debugging feed parsing issues**
- Keeps a record of the **original threat feed content**

---

3. count_ioc_types(iocs)

✅ **What it does:**

- Counts how many IOCs of each type exist (e.g., ip, url, domain, etc.)

📤 **Example:**

json

CopyEdit

```
{ "ip": 150, "url": 75 }
```

---

4. count_sources(iocs)

✅ **What it does:**

- Counts how many IOCs came from each source (e.g., spamhaus, digitalside)

📤 **Example:**

json

CopyEdit

{ "spamhaus": 50, "blocklist": 100 }

---

5. save_statistics(stats)

✅ **What it does:**

- Appends summary statistics of each run to:
  - data/processing_stats.json

✅ **Extra Features:**

- Adds a timestamp to each record
- Keeps **only the last 100 runs**

🧠 **Why:**

- Helps you monitor pipeline trends over time (e.g., number of IOCs processed, confidence stats)

🧾 **Example Output File: processed_iocs.json**

json

CopyEdit

```
{
 "metadata": {
  "last_updated": "2025-08-01T06:45:33.912Z",
  "total_iocs": 152,
  "ioc_types": { "ip": 100, "url": 52 },
  "sources": { "spamhaus": 70, "blocklist": 82 }
 },
 "iocs": [
  {
   "type": "ip",
```

"value": "8.8.8.8",
"confidence": 0.9,
"enrichment": { "country": "US", "org": "Google LLC" }
},
...
]
}

✅ Summary

Function Name	Role
save_processed_iocs	Saves final processed data and backups
save_raw_feed_data	Stores raw input for debugging or audits
count_ioc_types	Tallies types of IOCs (IP, URL, etc.)
count_sources	Tallies source feeds used for ingestion
save_statistics	Saves and rotates run-level metadata (IOC count, time, etc.)

File name: storage/load_data.py

This Python file handles **loading, restoring, and validating saved IOC data** for a threat intelligence system. It supports operations like reading processed IOCs from disk, loading metadata, restoring from backups, and verifying data integrity.

✅ Purpose:

To provide utility functions for:

- Loading saved IOCs and their metadata/statistics
- Recovering from the latest or specific backup
- Performing integrity checks on stored data

🔧 Function-by-Function Overview:

🔹 1. load_processed_iocs()

- Loads the main file: data/processed_iocs.json

- Returns both:
    - metadata
    - iocs (list of normalized IOC dictionaries)
- If file not found or error occurs, returns an empty structure with logs.

🔹 2. load_iocs_list()

- Returns only the list of IOCs (excluding metadata).

🔹 3. load_metadata()

- Returns only the metadata part (like total IOCs, sources, types, etc.).

🔹 4. load_statistics()

- Loads the statistics file: data/processing_stats.json
- Each entry logs a historical processing run.
- Returns a list of those statistics.

🔹 5. get_latest_backup()

- Looks in the data/ folder for files like processed_iocs_backup_<timestamp>.json
- Returns the **most recent** backup filename based on timestamp.

🔹 6. restore_from_backup(backup_path)

- Replaces the main processed IOC file with the contents of a backup.
- Useful for recovery after corruption or failure.

🔹 7. check_data_integrity()

Performs integrity checks on processed_iocs.json, including:

- Does the actual IOC count match metadata's total_iocs?
- Are required fields (id, value, type, source, confidence) present in each IOC?
- Are there any duplicate ids?

✔ **Returns:** an integrity report:

python

CopyEdit

```
{
  'is_valid': True/False,
  'total_iocs': 120,
  'issues': [ ... ],
  'metadata': { ... }
}
```

🧠 Why It's Important:

This module adds:

- **Reliability** → by checking for errors or mismatches
- **Recoverability** → by restoring from backups
- **Maintainability** → by letting developers access IOC data programmatically

---

📌 Summary:

This file provides **loading, backup recovery, and data validation** capabilities for a threat intelligence system that works with IOCs. It's the final layer in the data pipeline to ensure you can **read, trust, and recover your stored threat data** effectively.

🚀 Step: 4 API Interface (Python Flask)

- Run.py To run flask
- Start_pipeline.py for logging functionality
- Pipeline.py that connect flask
- App/template: frontend
- Routs.py: main function of flask
- Utils.py: functions that used in flask

File name: app/utiles.py

This Python file provides **utility functions** for a **Flask-based threat intelligence application**. Here's a breakdown of what each function does:

📃 Module Purpose:

python

CopyEdit

```
"""
```

Utility functions for the Flask application

"""

These are helper functions to **load**, **filter**, and **manage** Indicator of Compromise (IOC) data — typically for use in a backend service or dashboard.

🔧 Function Breakdown:

✅ load_iocs()

python

CopyEdit

```
def load_iocs() -> List[Dict]:
```

Purpose:
Loads a list of IOC (Indicator of Compromise) records from the JSON file at data/processed_iocs.json.

Details:

- Returns a list of IOC dictionaries.
- If the file doesn't exist → returns an empty list.
- Logs an error if something goes wrong.

Example output:

python

CopyEdit

```
[
  {"type": "ip", "value": "192.168.1.1", "source": "spamhaus", ...},
  {"type": "url", "value": "http://malicious.site", "source": "digitalside", ...}
]
```

✅ filter_iocs(...)

python

CopyEdit

```
def filter_iocs(iocs: List[Dict], ioc_type: Optional[str] = None,
                source: Optional[str] = None) -> List[Dict]:
```

Purpose:
Filters the list of IOCs by **type** (ip, url, etc.) and/or **source** (spamhaus, blocklist, etc.).

Parameters:

- ioc_type: Optional filter for the type of IOC.
- source: Optional filter for where the IOC came from.

Example:

python

CopyEdit

```
filtered = filter_iocs(iocs, ioc_type='url', source='blocklist')
```

Returns:
Only IOCs that match the filter conditions.

✅ ensure_data_directory()

python

CopyEdit

```
def ensure_data_directory():
```

Purpose:
Ensures that the required data directories (data/ and data/raw/) exist on the filesystem.

Use case:
Call this at app startup to make sure you don't get FileNotFoundError when saving data.

🧠 Summary:

Function	Role
load_iocs()	Loads IOC data from a JSON file
filter_iocs()	Filters IOC list based on type and/or source
ensure_data_directory()	Creates data/ and data/raw/ folders if missing

File name: app/routes.py

This Python file defines the **Flask routes (API endpoints)** for a **Threat Intelligence API**. Let's break it down route by route for clarity.

🔧 Purpose:

python

CopyEdit

```python
"""
Flask routes for threat intelligence API
"""
```

This file provides the **RESTful API interface** (and dashboard route) to:

- Fetch threat IOCs (Indicators of Compromise)
- Refresh the data
- Get stats
- Serve a basic web dashboard
- Do a health check

🔍 ROUTES EXPLAINED:

✅ GET /iocs

python

CopyEdit

```python
@main.route('/iocs', methods=['GET'])
```

Purpose: Return a list of IOCs (with optional filtering by type and source)

Query Parameters:

- type → e.g. 'ip' or 'url'
- source → e.g. 'blocklist', 'spamhaus', etc.

Uses:

- load_iocs() to read from JSON
- filter_iocs() to apply filters

Response:

json

CopyEdit

```json
{
  "success": true,
  "count": 32,
  "data": [ ... IOCs ... ]
```

}

✅ POST /refresh

python

CopyEdit

```
@main.route('/refresh', methods=['POST'])
```

Purpose: Triggers the **pipeline** to re-fetch, normalize, enrich, and classify IOCs.

Internally calls:

python

CopyEdit

```
result = run_pipeline()
```

Returns: success/failure message and how many IOCs were processed.

✅ GET /health

python

CopyEdit

```
@main.route('/health', methods=['GET'])
```

Purpose: Simple health check to verify the app is running.

Returns:

json

CopyEdit

```
{
  "status": "healthy",
  "service": "threat-intel-api"
}
```

✅ GET /

python

CopyEdit

```
@main.route('/', methods=['GET'])
```

Purpose: Renders a basic **HTML dashboard** (served using Jinja via render_template).

You’ll need a templates/index.html file for this to work.

✅ GET /api/stats

python

CopyEdit

```
@main.route('/api/stats', methods=['GET'])
```

Purpose: Returns statistical summary of IOCs by:

- Type (ip, url, etc.)
- Source (blocklist, etc.)
- Confidence:
    - **High**: > 0.8
    - **Medium**: 0.5 – 0.8
    - **Low**: < 0.5

Response Example:

json

CopyEdit

```json
{
  "success": true,
  "statistics": {
    "total_iocs": 100,
    "by_type": {"ip": 60, "url": 40},
    "by_source": {"spamhaus": 70, "digitalside": 30},
    "by_confidence": {"high": 50, "medium": 30, "low": 20}
  }
}
```

🧠 Summary Table:

Endpoint	Method	Purpose
/iocs	GET	Get IOCs, optionally filtered
/refresh	POST	Refresh and re-run IOC pipeline
/health	GET	Health check

Endpoint	Method	Purpose
/	GET	Serve a basic HTML dashboard
/api/stats	GET	Get stats on types, sources, and confidence