

STIX automation tool

Final deliverable in fulfilment of the final year project module

Authored by Huzaifah Machher

Supervised by Muhammed Ali Bingol

Word count: 10597

De Montfort University, Leicester

Contents

Acknowledgements	4
Abstract.....	4
1. Introduction	5
1.1 Background.....	5
1.2 Problem statement.....	5
1.3 Project motivation	5
1.4 Project aims	6
1.5 Scope and assumptions.....	7
2. System architecture overview	7
2.1 High level components	7
2.2 Backend – app.py	7
2.3 Frontend – .html and .js files.....	8
2.4 Dependencies	8
2.5 Security notes	8
3. Major components and implementation.....	8
3.1 Natural language processing and term extraction pipeline	8
3.2 Entity mapping	10
3.3 Backend API layer	11
3.4 Relationship modelling	13
3.5 Live suggestions	14
3.6 Frontend application logic – script.js	15
3.7 Results – results.html, results.js.....	16
3.8 Data assets	17
3.9 Styling and user experience (UX).....	19
4. Development lifecycle	22
4.1 Iteration 1 – NLP prototype	22
4.2 Iteration 2 – Full redesign	24
4.3 Iteration 3 (Final) – Cached term integration and target inference	26
4.4 Learning outcomes	27
5. Critical Analysis	28
5.1 Appraisal	28
5.2 Future enhancements from hindsight data	29
5.3 Analysis of approach	30
5.4 Unit testing.....	30

Table of figures

Figure 1. formatted JSON output.....	6
Figure 2. project hierarchy.....	7
Figure 3. logic for indicators of compromise extraction	9
Figure 4. logic for domain normalisation	9
Figure 5. logic for spaCy pipeline NER	9
Figure 6. extracted sightings from text under 3.1	9
Figure 7. JSON demonstrating mapping capabilities	10
Figure 8. fuzz.WRatio score visualised	11
Figure 9. live suggestion manifestation in app	12
Figure 10. object before conversion logic applied	12
Figure 11. object after conversion logic applied	12
Figure 12. displaying objects API call	13
Figure 13. built relationship.....	13
Figure 14. error message where src and dest are the same.....	14
Figure 15. logic pertaining to live suggestions	15
Figure 16. part logic for posting results and session storage	16
Figure 17. results page demonstration	17
Figure 18. terminal message for when terms are loaded.....	17
Figure 19. live suggestion for 'mimik'	18
Figure 20. excerpt from cache	18
Figure 21. logic pertaining to cache of terms loading, with fallback terms	18
Figure 22. landing page UI	20
Figure 23. main authoring page.....	21
Figure 24. results page	22
Figure 25. legacy UI implementation	24
Figure 26. inferred campaign name assignment	27
Figure 27. disclaimer for tool use within results.html	27

Acknowledgements

Completion of this module, and subsequently any other module would not have been possible without the unwavering support of my wife who has been the north star in a sea of uncertainty, her advice and reassurance lighted the path for me to successfully complete this project thus ending my studies as a Cyber Security undergraduate at De Montfort University. I would also like to express my gratitude to my parents for their lifelong care and firm support. Their constant encouragement and dedication are fundamental to my personal and professional growth, of which without, would not be possible.

I would also like to thank the teaching staff at De Montfort University for their invaluable expertise and insight into the topic of cyber security, throughout my time as an undergraduate student I have been both inspired and humbled by the vast amounts of knowledge possessed by these individuals. Their willingness to share has been a driving factor in the advancement of my academic success. Furthermore, I would like to thank Professor Nick Ayres for providing the idea for this project. His delivery of the Incident Response module covering the concept of STIX was most helpful to this project.

I am grateful to the OASIS STIX community for setting open standards and sharing reference implementations, including the stix2 library and validator, which guided correct modelling throughout this work. I also thank the many maintainers on GitHub whose efforts on spaCy, Flask, rapidfuzz, tldextract, and iocextract underpinned the extraction pipeline and application logic. FreeCodeCamp supported HTML, responsive CSS, and accessible interface patterns.

My friends, both old and new, have been a major support in my time at university. They have constantly provided me with fresh perspective and valuable information on relevant topics, whilst simultaneously offering a required distraction from study. Their companionship was a key motivator in difficult times during my years of study.

Abstract

Included in this report is the design and development lifecycle of a STIX automation tool which takes natural language user input and transforms it into actionable threat intelligence (TI) compliant with STIX 2.1 in JSON structure. The project addresses the increasingly important need for cyber threat intelligence modelling, and how this tool can help bridge the skill gap for individuals not familiar with the STIX requirements or JSON format.

The system combines natural language processing (NLP) parsing, contextual threat mapping, and structured entity creation for the purpose of TI modelling for a range of users. This includes security professionals and other analysts who want to quickly model cyber threats. Using a lightweight Flask backend, the tool integrates local threat intelligence term caches with spaCy NER for nouns, iocextract for robust Indicator of Compromise (IoCs) detection (including defanged formats), and tldextract for accurate domain parsing. RapidFuzz powers live suggestions and typo nudges, while the official stix2 Python library is used to construct SDOs/SCOs and SROs and to bundle outputs, almost irradicating manual assembly. On the frontend, plain JavaScript, along with HTML and basic CSS, manages user interactions and object creation, while also providing a straightforward button structure for ease of use. Additional client-side features include manual SRO assignment, real-time suggestions during input, and download of JSON contents.

The full development lifecycle is inclusive of user interface prototyping, and hardcoded models to fully dynamic object generation. Features to verify robustness such as data validation, dynamic threat mapping and SRO functionalities are discussed in detail with reference to code.

The tool is tested for its usability, ability to handle complexity and accuracy in modelling instances. To conclude, a critical reflection of the project highlights the iterations of the system, challenges and possible future implementation for the tool such as implementation of AI or dynamic addition of new cyber terms.

1. Introduction

1.1 Background

With the evolution of technology, attacks become increasingly sophisticated and prominent, as the cyber threat landscape widens, the need for cybersecurity tools that restore equilibrium becomes imperative. Threat intelligence sharing is crucial to combating cyber threat, when attacker techniques, tactics, and procedures (TTPs) can be identified, detection and response time is reduced, and the threat landscape is better managed.

STIX 2.1, managed by OASIS, provides a comprehensive environment for describing cyber threat instances such as malware, indicators, threat actors, and their relationships. However, manually creating STIX bundles often requires detailed knowledge of the STIX schema and tools like stix2 libraries or STIX editors. This project addresses this issue by bridging the skill gap, as the tool will allow users to input natural language threat intelligence and dynamically output STIX compliant structured threat intelligence in JSON format.

By introducing this tool, the technical barrier to using the STIX standard is lowered. With greater accessibility, threat intelligence sharing is encouraged. Promoting such behaviours better enhances the overall effectiveness of cyber defence.

1.2 Problem statement

Security professionals like analysts, responders and threat hunters often document findings in an unstructured, ununified and inconsistent way. While this method appears to be the fastest approach, it is counterproductive to automated reasoning, threat correlation, and rule-based security measures at scale. The lack of structure makes it hard to share intelligence between systems, hinders enrichment, and increases the chance of modelling errors when converting to STIX by hand.

1.3 Project motivation

As noted above, a widely adopted standard like STIX demands a technical grounding in its schema and the JSON language syntax. Where natural language descriptions of threat intelligence could look like this:

“On July 20, 2025, a phishing email impersonating Microsoft was sent to multiple corporate employees. The email contained a malicious link leading to a fake login page designed to steal credentials. The attackers used the domain login-micr0soft-support[.]com, registered just two days prior. The stolen credentials were later used to access internal systems and exfiltrate data. The campaign is suspected to be linked to the threat group APT29.”

A minimal STIX representation could include an Indicator an Identity, a Threat Actor and a Relationship like so:

```

1  {
2    "type": "bundle",
3    "id": "bundle--3f0a3b1f-6a30-4f9e-8b37-2b5f0b8a6c10",
4    "objects": [
5      {
6        "type": "campaign",
7        "spec_version": "2.1",
8        "id": "campaign--6b2a2d6a-3c57-4e0f-9d3a-2b5f9b1a2c44",
9        "created": "2025-08-14T00:00:00Z",
10       "modified": "2025-08-14T00:00:00Z",
11       "name": "Microsoft-themed credential phishing"
12     },
13     {
14       "type": "threat-actor",
15       "spec_version": "2.1",
16       "id": "threat-actor--2d4a7f0f-9f7e-4c4e-8b6a-1d3c2b7e9a55",
17       "created": "2025-08-14T00:00:00Z",
18       "modified": "2025-08-14T00:00:00Z",
19       "name": "APT29"
20     },
21     {
22       "type": "identity",
23       "spec_version": "2.1",
24       "id": "identity--f3c5b8b3-2a9f-4d7f-9c2e-8a1b5d7e4c66",
25       "created": "2025-08-14T00:00:00Z",
26       "modified": "2025-08-14T00:00:00Z",
27       "name": "Microsoft",
28       "identity_class": "organization"
29     },
30     {
31       "type": "identity",
32       "spec_version": "2.1",
33       "id": "identity--8e1d3c6b-5a7f-4d2e-9b3a-6c8d2f1e7b77",
34       "created": "2025-08-14T00:00:00Z",
35       "modified": "2025-08-14T00:00:00Z",
36       "name": "Multiple corporate employees",
37       "identity_class": "group"
38     },
39     {
40       "type": "attack-pattern",
41       "spec_version": "2.1",
42       "id": "attack-pattern--b6d1f2c3-7e8a-4f0b-9d2c-1a3b5e7f9c88",
43       "created": "2025-08-14T00:00:00Z",
44       "modified": "2025-08-14T00:00:00Z",
45       "name": "Phishing"
46     },
47     {
48       "type": "infrastructure",
49       "spec_version": "2.1",
50       "id": "infrastructure--1a2b3c4d-5e6f-4a7b-8c9d-0e1f2a3b4c59",
51       "created": "2025-08-14T00:00:00Z",
52       "modified": "2025-08-14T00:00:00Z",
53       "name": "Fake Microsoft login page"
54     },
55     {
56       "type": "indicator",
57       "spec_version": "2.1",
58       "id": "indicator--0a1b2c3d-4e5f-6789-abcd-0e1f2a3b4c6d",
59       "created": "2025-08-14T00:00:00Z",
60       "modified": "2025-08-14T00:00:00Z",
61       "indicator_types": ["malicious-activity"],
62       "pattern": "[domain-name:value = 'login-microsoft-support.com']",
63       "pattern_type": "stix",
64       "valid_from": "2025-07-20T00:00:00Z"
65     },
66     {
67       "type": "relationship",
68       "spec_version": "2.1",
69       "id": "relationship--c1d2e3f4-5a6b-4c7d-8e9f-0a1b2c3d4e5f",
70       "created": "2025-08-14T00:00:00Z",
71       "modified": "2025-08-14T00:00:00Z",
72       "relationship_type": "attributed-to",
73       "source_ref": "campaign--6b2a2d6a-3c57-4e0f-9d3a-2b5f9b1a2c44",
74       "target_ref": "threat-actor--2d4a7f0f-9f7e-4c4e-8b6a-1d3c2b7e9a55"
75     },
76   ]
77 }

```

Figure 1. formatted JSON output

As evidenced in fig. 1, natural language and JSON format STIX compliant entities display a significant disconnect between them, this is what the STIX automation tool aims to fix.

1.4 Project aims

Driven by its core motivations, the primary goal of this project is to develop a tool that accepts natural language threat descriptions, leverages NLP and specialist parsing to extract relevant threat modelling information and dynamically generates a STIX 2.1-compliant JSON bundle. This functionality is delivered through a highly interactive and intuitive web application. Given the well-defined scope, the tool should remain lightweight and user-friendly, with a strong emphasis on accurate entity extraction and relationship mapping.

1.5 Scope and assumptions

This section sets out the prerequisites and assumptions for the project's correct operation.

The tool is designed for English input, whilst other languages may be accepted, behaviour and accuracy are not guaranteed. Assisted by exhaustive caches of threat intelligence terms, the tool relies on this to cover a wide range of edge cases, however results are not guaranteed. Outputs are assistive not authoritative so users should review and adjust (where necessary) results on every cycle, especially named entities and their assigned types or relationships. Finally, the system is intended as an aid to bridge the technical gap between natural language notes and STIX 2.1, not as a fully automated threat intelligence modelling solution.

2. System architecture overview

The STIX automation tool adopts a modular architecture comprised of a frontend user interface (UI) for authoring and review by user, and a Flask based backend application to handle parsing, entity extraction, threat intelligence data mapping and ultimately STIX2.1 bundle generation. A typical use case will follow this flow:

1. User enters plain text threat intel
2. User may choose to optionally defines relationships via drop down menu
3. "Generate STIX" calls the backend
4. Backend extracts entities, maps to STIX objects and relationships, and returns a bundle
5. Results page shows JSON with Copy, Download, and optional Edit buttons

2.1 High level components

	Component	Use
Front end	Landing.html	Overview of tool
	Main.html	Text area for user input and relationship modelling, features a live suggestion box for user text and the option to generate
	Results.html	Display for output JSON as a result of backend logic
	Script.js	Input handling, debounce for suggestions, relationship modelling management and fetch to backend
	Results.js	Displays bundle and facilitates export capabilities
	Styles.css	User interface properties
Backend	App.py	Entity recognition, extraction and mapping logic
	Data assets	Term caches for entity recognition and assignment

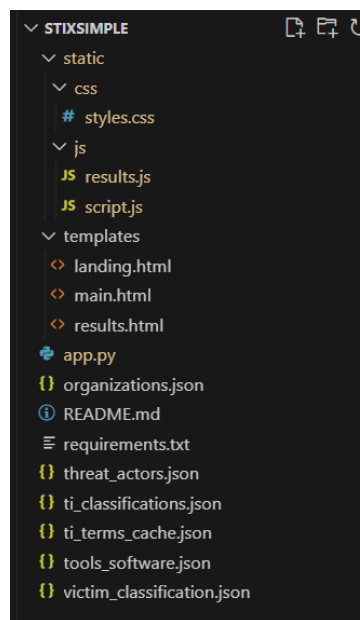


Figure 2. project hierarchy

2.2 Backend – app.py

- **Live suggestions:** Fuzzy matching techniques compare tokens against cached threat intelligence terms and returns top ranking matches.
- **Object discovery:** Quick pass to stage extracted entities for the relationship building model.
- **Conversion:** validates input, extracts entities, maps extracted entities to respective STIX defined fields ensuring compliance. Also applies user defined relationships and returns complete STIX bundle in JSON.

2.3 Frontend – .html and .js files

- **Authoring:** main.html features text area with debounced suggestions, features relationship building component using extracted entities.
- **Conversion and review:** ‘Generate STIX’ calls backend logic to model threat intelligence, results are shown on results.html in formatted JSON with options to copy, download, edit and return to main.html.

2.4 Dependencies

- **Caches:** a repository of an extensive list of threat intelligence terms including proper nouns, common terms and categories formatted against MITRE ATT&CK framework, manually extendable.
- **Technologies:** Python 3.11.9, Flask development server, spaCy en_core_web_sm at runtime.

2.5 Security notes

- **Storage:** Session storage preferred over server-side persistence.
- **Deployment:** CORS enabled, debug disabled, lightweight model adopting an essential only security position.

3. Major components and implementation

This section details the critical architectural and functional components of the STIX automation tool, inclusive of design reasoning, challenges faced and highlights from the codebase. Each subcomponent has a modular design to accommodate for extensibility.

3.1 Natural language processing and term extraction pipeline

Problem

Natural language and unstructured threat intelligence has an inherent potential for misinterpretation and loss of context. When drafting threat intelligence, observables are often defanged and names for individuals, groups, organisations and locations are subject to variation. Basic entity extraction algorithms whilst lightweight may not account for these edge cases, however, large natural language processing models can be slow and complex. A system should be able to find a balance between handling edge cases whilst keeping overhead and processing time to a respectable limit. Misinterpretation of proper nouns or other extractions may skew the end STIX results, making the information obsolete. This component provides a solution for maintaining the delicate balance by accurately and efficiently extracting entities from natural language text, so the STIX mapping function (discussed later) has accurate input for modelling purposes.

Objective

The pipeline aims to maximise identification of common indicators of compromise (IOCs) such as IPs, URLs, emails, etc., whilst also capturing named entities like people, organisations or locations. This system should stage clean and deduplicated entities for mapping later, without over interpreting ambiguous text. Responsiveness will be critical.

Implementation

The extraction function is called inside app.py by /api/get-objects and /api/convert. It runs spaCy’s Named Entity Recognition (NER) to identify proper nouns i.e., people, organisations, locations, etc, while Pythons inbuilt library iocextract is used to pull IOCs and refang them (if needed) whilst tldextract is used to canonicalize domains, all from user text.

IOC extraction using iocextract

```
197 | # Extract IOCs using iocextract library
198 | entities["ips"] = list(iocextract.extract_ips(text, refang=True))
199 | entities["urls"] = list(iocextract.extract_urls(text, refang=True))
200 | entities["emails"] = list(iocextract.extract_emails(text, refang=True))
201 |
202 | # Clean up email addresses that look like user@IP
203 | clean_emails = []
204 | for em in entities["emails"]:
205 |     dom = em.split("@")[-1]
206 |     # Skip emails with IP addresses as domains
207 |     if re.fullmatch(r"\d{1,3}(\.\d{1,3}){3}", dom):
208 |         continue
209 |     clean_emails.append(em)
210 | entities["emails"] = clean_emails
211 |
212 | entities["hashes"] = list(iocextract.extract_hashes(text))
```

Figure 3. logic for indicators of compromise extraction

Domain normalisation using tldextract

```
219 | # Extract domain names from URLs
220 | for url in entities["urls"]:
221 |     extracted = tldextract.extract(url)
222 |     if extracted.domain and extracted.suffix:
223 |         domain = f"{extracted.domain}.{extracted.suffix}"
224 |         if extracted.subdomain:
225 |             domain = f"{extracted.subdomain}.{domain}"
226 |         if domain not in entities["domains"]:
227 |             entities["domains"].append(domain)
```

Figure 4. logic for domain normalisation

NER for name and location extraction by spaCy

```
308 | # Use spaCy NER for organization and location detection
309 | doc = nlp(text)
310 | for ent in doc.ents:
311 |     if ent.label_ == "ORG":
312 |         ent_text = ent.text.strip()
313 |         ent_lower = ent.text.lower()
314 |         # Filter out generic organizational terms
315 |         skip = ["hr", "it", "department", "team", "group", "security", "admin", "support"]
316 |         if any(s in ent_lower for s in skip) or len(ent_text.split()) < 2:
317 |             pass
318 |         else:
319 |             # Only add if not already classified
320 |             if ent_lower not in THREAT_ACTORS_MAP and ent_lower not in TOOLS_MAP and ent_lower not in ORGS_MAP:
321 |                 if 3 < len(ent_text) < 50:
322 |                     entities["identities"].add(ent_text)
323 |             elif ent.label_ in ("GPE", "LOC"):
324 |                 loc = ent.text.strip()
325 |                 if 2 <= len(loc) <= 80:
326 |                     entities["locations"].add(loc)
```

Figure 5. logic for spaCy pipeline NER

For illustrative purposes user inputs: "On 20 July 2025, a phishing email impersonating Microsoft lured ACME Corp staff in London to [hxxps://login-micr0soft-support\[.\]com](https://login-microsoft-support[.]com). Suspected actor: APT29." The system parses through entities as:

```
1 | {
2 |     "ips": [],
3 |     "urls": ["https://login-microsoft-support.com"],
4 |     "domains": ["login-microsoft-support.com"],
5 |     "emails": [],
6 |     "hashes": [],
7 |     "named": [
8 |         ["Microsoft", "ORG"],
9 |         ["ACME Corp", "ORG"],
10 |        ["London", "GPE"],
11 |        ["APT29", "ORG"]
12 |     ]
13 | }
```

Figure 6. extracted sightings from text under 3.1

Rationale

Importing spaCy's NER only, keeps requirements minimal and reduces overhead whilst returning the exact function required, whilst iocextracts defang and refang capabilities are consistent with how information is presented in reports.

In app.py, the model is loaded once, pipes not serving NER are disabled, outputs are normalised and deduplicating as the final guard ensures minimal error margins and reduced latency. Choosing this approach in place of training a custom model ensures the system is lightweight and true to scope.

3.2 Entity mapping

Problem

Extracted entities from text must conform to STIX2.1 schema to produce any actionable output, the fragile nature of JSON coupled with a strict STIX schema means a misplaced ',', missing required field, or wrongly formatted syntax breaks interoperability.

Objective

The system should define rules which are deterministic for all extracted indicators and entities to avoid instances where it must make unmeaningful assumptions. The system should ensure all sightings from source text are transformed into fully compliant STIX2.1 objects accurately. This conversion process should only assign to field and set relationships where clearly justified by available evidence, either from context or cross reference from cache files (discussed later). Ambiguous or unrelated to threat intelligence text should not be over interpreted as doing so risks producing misleading or invalid STIX data. The system must demonstrate the ability to map sightings to relevant fields whilst filtering out nonrelevant data.

Implementation

In app.py, mapping functions create stix2 objects by construction, from an example, similar to the previous one, identified object will be created as so:

```
{
  "id": "bundle---c7fbbbd6-e43e-4020-ad13-8162707448f5",
  "objects": [
    {
      "created": "2025-08-12T22:35:39.115492Z",
      "id": "indicator---a1d27df7-fd97-475a-982e-17558a0a64e0",
      "labels": [
        "malicious-activity"
      ],
      "modified": "2025-08-12T22:35:39.115492Z",
      "name": "Domain: login-microsoft-support.com",
      "pattern": "[domain-name:value = 'login-microsoft-support.com']",
      "pattern_type": "stix",
      "pattern_version": "2.1",
      "spec_version": "2.1",
      "type": "indicator",
      "valid_from": "2025-08-12T22:35:39.115492Z"
    },
    {
      "created": "2025-08-12T22:35:39.115492Z",
      "id": "indicator---627e5217-8879-454a-adc3-ad2c167fefb3",
      "labels": [
        "malicious-activity"
      ],
      "modified": "2025-08-12T22:35:39.115492Z",
      "name": "URL: http://login-microsoft-support.com",
      "pattern": "[url:value = 'http://login-microsoft-support.com']",
      "pattern_type": "stix",
      "pattern_version": "2.1",
      "spec_version": "2.1",
      "type": "indicator",
      "valid_from": "2025-08-12T22:35:39.115492Z"
    }
  ]
}
```

Figure 7. JSON demonstrating mapping capabilities

From the extracted payload, domain is observed as an indicator, with the rest of the payload mapping to their respective fields i.e., ThreatActor(name="APT29", threat_actor_types=["nation-state"]).

Rationale

Using the stix2 import which is designed specifically for this purpose ensures there is no deviation from schema, enforcing required information like IDs and timestamps. Being the official and actively maintained implementation provided by the OASIS standards community (same group which governs STIX itself) an accurate and trustworthy output is better guaranteed, ensuring high quality reporting and better relationship building.

Choosing to import Python library stix2 rather than fashioning entirely custom entity mapping logic has a significant impact on development efficiency and maintains a higher accuracy and guarantee of results, recognised libraries also provide long term maintainability where custom logic may prove to be difficult to refactor if necessary.

3.3 Backend API layer

Problem

The UI requires that the suggestions are made in real time, a consistently updated entity list for relationship modelling and a definitive text to bundle conversion process. Each of these requirements must leverage small, predictable payloads as to maximise speeds.

Objective

To fulfil this requirement, script.js must be provided with three routes with well defined purposes so that it can orchestrate user interactions without client-side complexity.

Implementation

The routes previously mentioned have defined inputs and outputs:

- /api/live-suggestions – handles live suggestions
- /api/get-objects – take text and returns normalised sightings without creating any STIX objects
- /api/convert – accumulates all extractions and user defined relationships, then returns a bundle which is subjected to validation

POST api/live-suggestions expects the current token and returns best matches from the caches of threat intelligence using RapidFuzz – another instance of an imported Python library, reducing code development overhead.

Feature *process.extract(...)* from RapidFuzz found in app.py compares *current_word* to every string in *choices*, computes a similarity score for each and returns the best matches determined by highest score. *scorer=fuzz.WRatio* dictates which similarity function to use with *WRatio* being a general-purpose scorer that combines different string comparison strategies, this is particularly useful when there are typos present in the current token. *limit=5* defines how many best matches should be used.

For example, if `{ "current_word": "ransomwar" }`, the response featuring fuzzy logic will manifest:

```
1  { "suggestions": [  
2    { "word": "ransomware", "score": 96},  
3    { "word": "other_text", "score": <96}  
4  ] }
```

Figure 8. fuzz.WRatio score visualised

Each tuple appears as *(matched_choice, index_in_choices)* in the backend logic where *matched_choice* is the suggested word, *score* which determines similarity and *index_in_choice* defines where the suggestion ranks. The UI will only display the suitable matches to a predefined amount of 5 choices. When a suggested word is selected by user, script.js replaces the current word with the suggestion, once interacted with or if the word is removed, the suggestion disappears, ready for the next iteration.

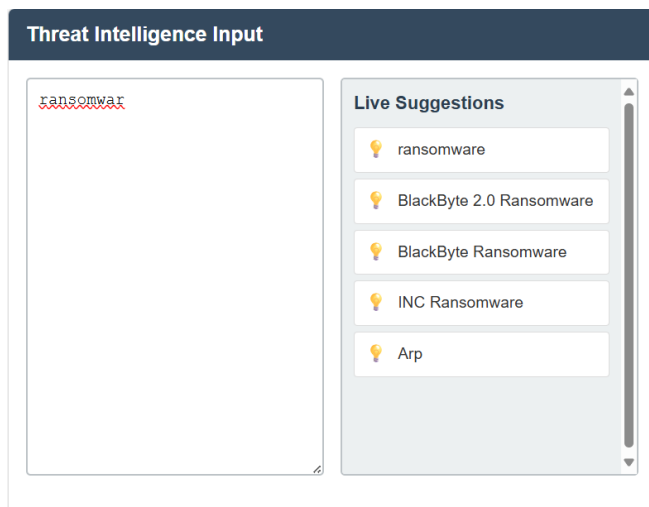


Figure 9. live suggestion manifestation in app

/api/get-objects is called by script.js when user clicks “Generate STIX” on main.html. User input is parsed and validate using simple guards, if the input is insufficient, defined either by a lack of extractable information or an unsatisfactory character length, the algorithm returns network code 400. At this stage there is no STIX construction, sightings are simply parsed to offer the relationship modelling function content. Script.js also calls *SmartSTIXConverter.extract_all_entities(text)*, which combines functionalities of various Python libraries mentioned before to extract relevant threat intelligence data.

/api/convert manifests itself on creation of the final STIX bundle, after text has been validated, a full extraction is run, data is mapped to STIX 2.1 objects and user defined relationships are merged, returning the final bundle. This function runs *extract_all_entities(text)* (same pipeline used in */api/get-objects*) and calls *create_stix_bundle(entities, relationships)* to build objects via import stix2 constructors. For instance, where a request could look like:

```

1  {
2    "text": "APT29 targeted ACME staff using login-microsoft-support[.]com.",
3    "relationships": [
4      { "source": "APT29", "type": "targets", "target": "ACME Corp" }
5    ]
6  }

```

Figure 10. object before conversion logic applied

The response would look like:

```

7  {
8    "stix": { "type": "bundle", "spec_version": "2.1", "objects": [ /* ... */ ],
9    "summary": { "indicators": 1, "threat_actors": 1, "identities": 1, "relationships": 1 }
10  }
11  }
12  }

```

Figure 11. object after conversion logic applied

stix2 facilitates bundling of objects and ensures compliance with the STIX standard when creating STIX objects i.e., STIX Domain Objects (SDOs), STIX Cyber-observable Objects (SCOs), STIX Relationship Objects (SROs). It also has the capability to dynamically populate required fields where applicable and create IDs and timestamps. This comprehensive solution to building STIX data ensures a low error rate and eliminates need for a bespoke logic.

RapidFuzz is the newer and faster alternative to its predecessor FuzzyWuzzy. Its function is to compare strings and give them a rating on similarity from 0 – 100. With underlying complex algorithms, it is a comprehensive solution for live suggestions where custom logic may fail to provide similar results.

Rationale

Importing these libraries whether custom or external avoids reimplementing fixes for hard problems that are renowned for edge cases. Choosing this over hand-crafted regex/NLP and JSON building algorithms provides a more optimised approach. Other styles of implementation will be slower to build, fragile and more prone to missing valid entities or even hallucinating them.

3.4 Relationship modelling

Problem

Relational data is a valid STIX modelling process, however, as with all STIX Objects it must conform to the schema outlined in the STIX documentation. Relational Objects must be of fixed type, with a dozen options to choose from, a typical user may naturally not remember them all. Without reference to the documentation an unrestricted natural language manual approach to creating SROs can lead to invalid links or failed creation entirely, consequentially slowing down the threat intelligence modelling process.

Objective

Provide a focused drop-down menu system which stages objects found only in the current user text. This simple workflow enables users to quickly and easily assign relational data to STIX objects and prevents any invalid links being made by prepopulating the SRO types.

Implementation

At main.html, clicking “Add Relationship” triggers the drop-down menu form. On click script.js simultaneously calls on /api/get-objects to populate all sections within the form including source, type and target. Suppose the user selects source as "APT29", type as "targets", target as "ACME Corp". script.js then appends:

```
relationships.push({ source: "APT29", type: "targets", target: "ACME Corp" });
```

```
281 |  * Loads available objects from the current text for relationship building
282 |  */
283 |  async loadAvailableObjects() {
284 |    try {
285 |      // Request object extraction from server
286 |      const response = await fetch("/api/get-objects", {
287 |        method: "POST",
288 |        headers: { "Content-Type": "application/json" },
289 |        body: JSON.stringify({ text: this.inputArea.value.trim() }),
290 |      });
291 |
292 |      const data = await response.json();
293 |      this.availableObjects = data.objects || [];
294 |    } catch (error) {
295 |      this.showMessage("Error loading objects", "error");
296 |    }
297 |  }
298 | }
```

Figure 12. displaying objects API call

to an in-memory list rendered beneath the editor. No data is sent to the server at this stage; the full list is posted with the text to /api/convert when the user clicks “Generate STIX”. After each instance of creating an SRO, its data is displayed in a plain text list for user attention, relationships can also be deallocated using the “Remove” button displayed next to each created relationship.


 Domain: login-micr0soft-support.com detects URL:
http://login-micr0soft-support.com Remove

Figure 13. built relationship

Rationale

Heavy importance is placed on simplicity of use, using the relationship modelling drop-down menu users can quickly and efficiently add context to threat intelligence without needing to identify plain text entities or relational choices manually.

Certain scenarios will trigger error messages from script.js, users may not map an SDO to itself and this will be made clear on attempt. Users must also populate all fields before initialising an SRO, failure will prompt a different error message. Well informed error messages, simple UI components and clearly labelled sections ensure ease of use.

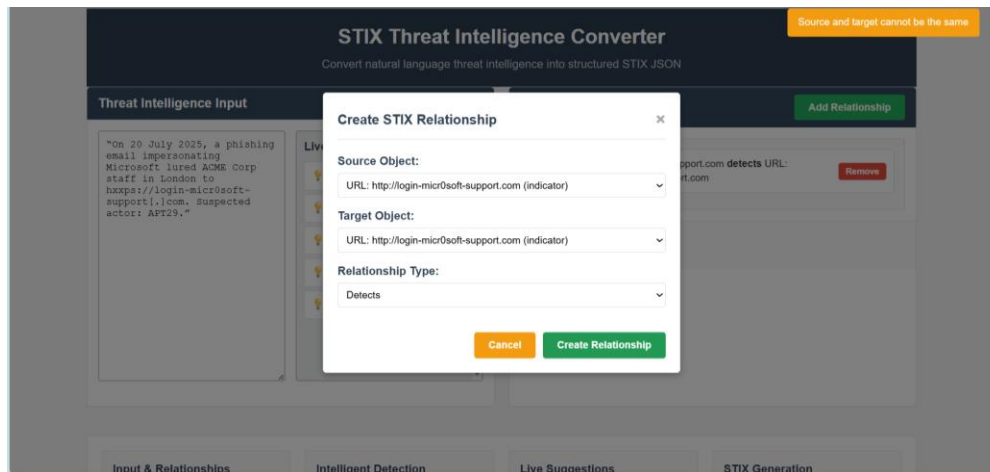


Figure 14. error message where src and dest are the same

3.5 Live suggestions

Problem

Natural language user input is susceptible to human error, without any guard rails typos and inconsistent wording reduce successful extractions. For example, an instance of “malwear” instead of “malware” in text may not match cached terms which the system calls on to provide suggestions. On the contrary, an overly eager autocomplete would suggest unlikely matches.

Objective

Offer gentle vocabulary hints tied to the current token under the cursor, with a ‘click to accept’ behaviour that doesn’t disrupt flow, providing only closely matched alternatives without influencing user input.

Implementation

As the user types in main.html, script.js starts a short debounce and posts the current token only to the beforementioned /api/live-suggestions. When input logic looks like `{“current_word”:“malwear”}`, app.py refers to the cumulative cache of terms under `ALL_TERMS`, with assistance from RapidFuzz the system returns a predefined maximum of 5 highest ranking words, determined by fuzzy logic. For this example, “malwear” should return “malware” as its top-ranking suggestion, clicking on the suggestion will replace the token under the cursor by logic:

```

172  /**
173   * Replaces the current word with the selected suggestion
174   * @param {String} original - The original word being replaced
175   * @param {String} suggested - The suggested replacement word
176   */
177  applySuggestion(original, suggested) {
178    const text = this.inputArea.value;
179    const cursorPosition = this.inputArea.selectionStart;
180
181    // Calculate the position of the current word
182    const beforeCursor = text.substring(0, cursorPosition);
183    const afterCursor = text.substring(cursorPosition);
184    const words = beforeCursor.split(/\s+/);
185    const currentWord = words[words.length - 1];
186
187    // Replace the current word with the suggestion
188    const beforeWord = text.substring(0, cursorPosition - currentWord.length);
189    const newText = beforeWord + suggested + afterCursor;
190
191    this.inputArea.value = newText;
192
193    // Move cursor to end of the suggested word
194    const newCursorPosition = beforeWord.length + suggested.length;
195    this.inputArea.setSelectionRange(newCursorPosition, newCursorPosition);
196    this.inputArea.focus();
197
198    // Reset suggestions panel
199    this.showDefaultSuggestions();
200  }
201

```

Figure 15. logic pertaining to live suggestions

After a token expires, so does its suggestion meaning no visual cluttering of UI.

Rationale

The live suggestion feature focuses solely on the current token under the cursor and does not concern itself with full text context. This approach prevents large volumes of text from being sent back and forth and avoids generating oversized suggestion lists. Since live suggestions must keep pace with user input, minimising the payload size for each suggestion helps reduce lag and ensures the system can operate smoothly and responsively.

The matching of user input speed is imperative to usability and user flow, as overly predictive text may cause confusion when typing, debouncing implementations ensure that suggestions are only provided when user pauses, keeping jitter to a minimum. The function is purely suggestive, meaning no rewriting without user approval.

3.6 Frontend application logic – script.js

Problem

User text typing, live suggestions, relationship modelling, and the final conversion call are part of the main three components that serve main.html, of which can interfere with one and other if they are not carefully coordinated. Without safeguards race conditions can occur where an older, slower network response overwrites a newer one. This can result in outdated suggestions, a previous draft of objects being populated in the relationship drop-down, and double submissions if users click Generate twice while the first request is still being fulfilled. Error feedback can also fragment if each feature reports problems differently, leaving users unsure what failed or how to recover.

Objective

Make interactions predictable and recoverable. Concretely, defer network calls with a small debounce so requests are not made to the server more frequently than is required, to do this the system should ensure only the latest async response can update the UI and centralise error handling so all failures render the same way. The system must set out rules of use, disabling functions before a defined condition is met will mitigate user error and direct flow, in the same vein all interactions should cause reaction from the system, always keeping the user well informed i.e., clear error messages, subtle changing of UI appearances.

Implementation

Script.js attaches listeners to the various components of the main.html file, namely the text area where users type, the relationship modelling area and all buttons. When the user opens the relationship window, it fetches /api/get-objects, normalises the payload into dropdown options, and enforces simple client-side validation before queuing the relationship. On Generate, it checks a minimum word count, posts the text and the queued relationships to /api/convert, and, on success, saves the bundle into sessionStorage under a key like "stixBundle" before navigating to results.html.

```
236 |     try {
237 |       // Send conversion request to server
238 |       const response = await fetch("/api/convert", {
239 |         method: "POST",
240 |         headers: { "Content-Type": "application/json" },
241 |         body: JSON.stringify({
242 |           text: text,
243 |           relationships: this.relationships,
244 |         }),
245 |       });
246 |
247 |       const data = await response.json();
248 |
249 |       if (data.stix) {
250 |         // Store the STIX output in session storage
251 |         sessionStorage.setItem("stixOutput", JSON.stringify(data.stix));
252 |         // Navigate to results page
253 |         window.location.href = "/results";

```

Figure 16. part logic for posting results and session storage

Rationale

Script.js is a lightweight and simple solution for orchestrating the typical behaviours from the UI, each component is meticulously handled by the .js file to ensure smooth operation and guided user flow. All network calls set consistent headers and parse JSON through a single helper where action buttons are disabled during requests, this is to prevent double submissions. Also toasts share styling via a small utility, so user feedback is uniform.

3.7 Results – results.html, results.js

Problem

After STIX data has been converted, the results must be displayed in a clear and accurate JSON format that is fully compliant with the STIX schema. Users should be able to review this output before exporting, ensuring they can verify its correctness. If discrepancies arise, whether due to system logic or user error, the workflow should not require starting over from scratch. Instead, an editing feature should be provided, allowing the user to manually adjust the generated output. Finally, users must be offered straightforward options for exporting the data in their preferred format.

Objective

Present a single, readable JSON text with native export functionality, with the optional capability to edit output where necessary.

Implementation

On render, results.js digests the bundle from sessionStorage and displays it's formatted contents compliant with JSON and STIX into the text area featured in results.html, accompanied by three related follow up actions:

- Copy – Calls the Clipboard API with an execCommand archetype.
- Download – Creates a Blob and a temporary object URL to save *stix-bundle-<timestamp>.json*
- Edit – Toggles off readonly and updates button state.

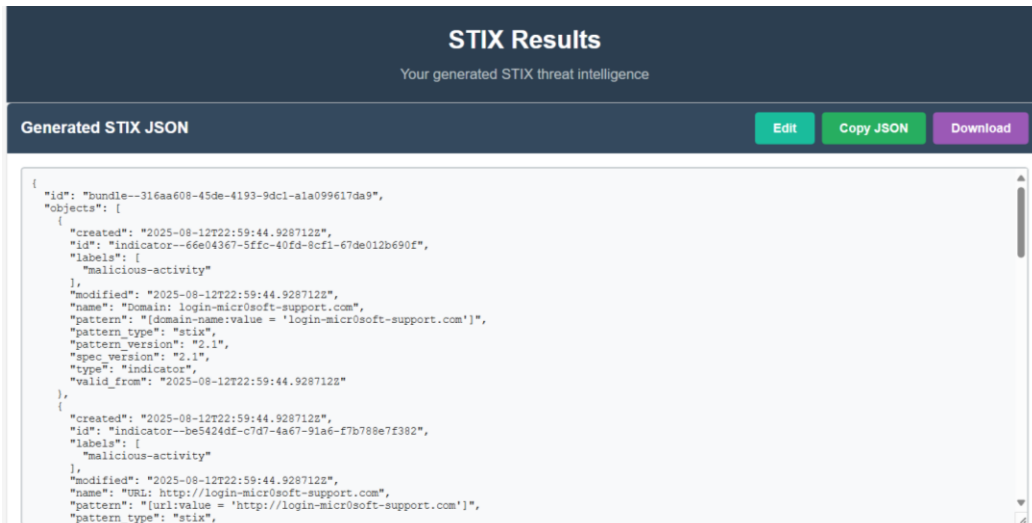


Figure 17. results page demonstration

Rationale

Whilst displaying individual STIX objects may reduce overhead and produce easier digestible information for users, displaying a complete JSON render of the previously created STIX bundle mirrors the format which will be used by tools to ingest this information (a single artifact).

Editing can be done manually without the need for revisiting the modelling process, users can directly interact with results for efficient editing. Output is locked by default until editing is enabled to account for any accidental actions from user, preserving the STIX data for later use. Results.js always revokes object URLs to avoid leaks. The file name includes a timestamp to prevent accidental overwrites.

3.8 Data assets

Problem

SpaCy's NER although used for proper nouns, is not aware of threat intelligence terms by default, this creates a significant problem for comprehensive entity extraction. Without aid, mapping would suffer from missed context from user input, resulting in a less complete STIX bundles.

Objective

Provide a list of curated threat intelligence terms that can improve recall in suggestions, and aid mapping, while keeping the scope for more terms to be added later for comprehensiveness.

Implementation

On boot, app.py loads:

- ti_terms_cache.json – Extensive threat intelligence dictionary
- threat_actors.json – List of known threat actors
- organizations.json – List of organisations so the system logic can differentiate between adversary and victim
- tools_software.json – List of known tools
- victim_classification.json – list of victim identifiers

```
Starting STIX converter with 2165 threat intelligence terms
Libraries loaded: spaCy, iocextract, tldextract, rapidfuzz, stix2
```

Figure 18. terminal message for when terms are loaded

For example, if “Mimikatz” appears in *tools_software.json*, then typing “mimik...” will prompt a suggestion, and a strong exact match during mapping can seed a Tool object in */api/convert*. This is especially helpful to the system where extremely foreign entities are either missed or identified but not categorised.

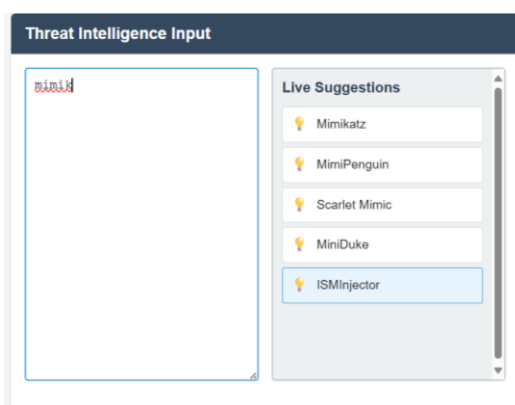


Figure 19. live suggestion for 'mimik'

Rationale

A library of cached terms implementation as opposed to a machine learning (ML) implementation drastically reduces the complexity and computation requirements of the proposed challenge. The knowledge base can be manually expanded effortlessly by scraping terms from other dictionaries, without any need for complex model retraining. Although susceptible to missed context, the implementation provides an almost complete solution, balancing effort and results appropriately.

The loader in *app.py* ensures all terms retain their original forms whilst also storing their lowercase counterparts, merges duplicate deterministically and tolerates missing files with warnings and provides hardcoded fall-back data sets.

```

1  {} ti_terms_cache.json > ...
2  {
3    "hdoor": "hdoor",
4    "trickbot": "trickbot",
5    "cdoos": "cdoos",
6    "powerduke": "powerduke",
7    "ekans": "EKANS",
8    "blindingcan": "BLINDINGCAN",
9    "ninja": "Ninja",
10   "pikabot": "Pikabot",
11   "warp": "warp",
12   "resession": "Resession",
13   "spark": "Spark",
14   "quiesleive": "Quiesleive",
15   "synack": "Synack",
16   "bumblebee": "Bumblebee",
17   "murkytop": "MURKYTOP",
18   "acidrain": "Acidrain",
19   "griffon": "Griffon",
20   "exaramelforwindows": "Exaramel for Windows",
21   "amadey": "Amadey",
22   "dumbledoth": "dumbledoth"
23 }

```

Figure 20. excerpt from cache

```

42 # Load threat intelligence data from cache files
43 TI_TERMS = load_json_safe("ti_terms_cache.json", [])
44 TI_CLASS = load_json_safe("ti_classifications.json", [])
45 THREAT_ACTORS = load_json_safe("threat_actors.json", [])
46 ORGS = load_json_safe("organizations.json", [])
47 TOOLS = load_json_safe("tools_software.json", [])
48 VICTIM_TERMS = load_json_safe("victim_classification.json", [])
49
50 # Create lookup dictionaries for faster searching
51 THREAT_ACTORS_MAP = {str(x).lower(): x for x in (THREAT_ACTORS or [])}
52 ORGS_MAP = {str(x).lower(): x for x in (ORGS or [])}
53 TOOLS_MAP = {str(x).lower(): x for x in (TOOLS or [])}
54 VICTIM_TERMS_MAP = {str(x).lower(): x for x in (VICTIM_TERMS or [])}
55
56 # Combine all terms for the suggestion feature
57 ALL_TERMS = list(TI_TERMS.values()) + THREAT_ACTORS + ORGS + TOOLS
58
59 # Define generic terms for fallback detection
60 GENERIC_TA_TERMS = ["adversary", "attacker", "attackers", "threat actor", "threat actors", "apt"]
61 GENERIC_TA_NAME = "Adversary"
62
63 GENERIC_MALWARE_TERMS = ["malware", "ransomware", "trojan", "backdoor", "wiper", "worm", "virus", "spyware"]
64 GENERIC_MALWARE_NAME = "Unknown Malware"

```

Figure 21. logic pertaining to cache of terms loading, with fallback terms

3.9 Styling and user experience (UX)

Problem

Disorganised or inconsistent UIs pull attention away from the purposed task. Users benefit from a compliant UI so that concentration can be applied to the end goal rather than other tasks which should typically be easy. A messy UI does not inspire confidence in users, potentially manifesting as a downturn in usage due to mistrust, and without a clear and informative display users may not be familiar with the tool's functionality or technologies.

Objective

The UI should make navigation clear and easy whilst keeping the user informed where possible. A restrained and responsive design keeps controls predictable and readable on common mediums. Components should not appear distorted when viewing in different modes i.e., full screen or window.

Implementation and rationale

The system is comprised of three distinct html files, each serving a well-defined purpose backed by a stylistic implementation suited to its use. The files are:

- landing.html – Entry point for users
- main.html – Presents the main functionality of the tool
- results.html – Final destination for users

landing.html acts as a starting point and introduction to prospective users for the STIX automation tool, with reference to styles.css for design, it demonstrates a clean and responsive design in the likeness of modern UIs. It defines:

- What the tool is – Its title and subtitle in *.landing-header*
- What the tool does – The hero section outlines key features like NLP powered analysis, real time suggestions
- How to get started – A call-to-action style “continue” button that links to /main.html
- Educational context – An “about this project” section disclaiming its experimental origins
- Informational content – Section with key features

styles.css defines:

- Responsiveness – On large screens, content is arranged in grids whilst on mobile elements stack vertically.
- Consistent styling – UI components like buttons, headers and sections use a harmonic colouring scheme to reduce visual noise and present a professional appearance.
- Content highlighting – Any important information is designed to stand out, drawing user attention through visual cues.

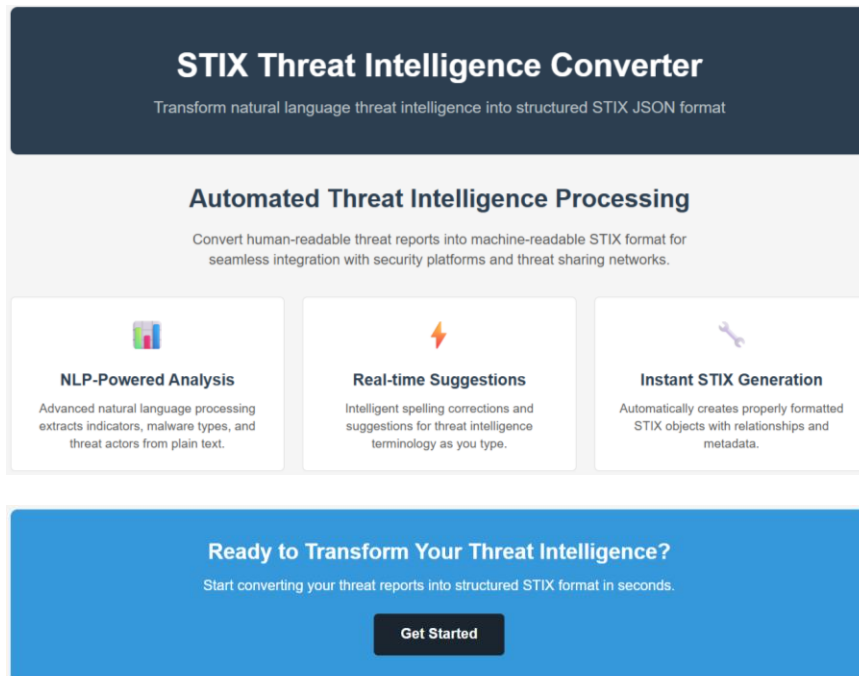


Figure 22. landing page UI

main.html presents the core operational display as the next logical step in design influenced user flow. It presents:

- Intelligent detection – Live suggestions in context.
- Relationship building – Simplistic drop-down menu for quick and non-taxing relationship modelling.
- Minimal distraction – Styles keep the user in a focused environment true to purpose.

styles.css defines:

- Layout – *.main-container* encapsulates the entire UI, maintaining a clear grid and predictable structure whilst *.io-wrapper* splits into two compartmentalisations, the input section featuring text area & suggestions, and the relationship section.
- Consistency – All buttons of each grouping share similar stylistic properties e.g., size, fonts, hover states, making controls feel predictable for users.
- Content focus – A large text area for input dominates half of the space, ensuring a feeling of abundance for users.
- Drop down menu – On click, the drop-down menu dominates all the space and simultaneously dims background to enhance focus on task.

STIX Threat Intelligence Converter
Convert natural language threat intelligence into structured STIX JSON

Threat Intelligence Input

Enter threat intelligence in plain English...

Live Suggestions

Type threat intelligence terms to see suggestions...

STIX Relationships

Add Relationship

No relationships created yet. Click "Add Relationship" to create one.

Generate STIX Clear

Input & Relationships
Enter threat intelligence and add relationships between detected objects on the same page for a streamlined workflow.

Intelligent Detection
Automatically detects hashes, IPs, domains, emails, malware types, attack patterns, and threat actors.

Live Suggestions
Real-time word suggestions based on threat intelligence vocabulary as you type.

STIX Generation
Generate compliant STIX JSON with proper object relationships and validation results.

Figure 23. main authoring page

results.html presents the results following STIX generation. It demonstrates:

- Output – A read only display of large monospace text for easy scanning and syntax clarity.
- Button facilitated actions – Ability to edit and export data both as a download or copy for clear navigation.
- Information panel – Content describing functions displayed on screen for clarity of use.

styles.css defines:

- Layout – *.main-container* maintains consistent margins and spacing with other html files, *.results-wrapper* centres the output with padding all round so JSON feels uncluttered. *.results-header* groups and features the title and action buttons designed for immediate recognition.
- Readability – JSON output uses a monospace font and generous line height which is important for easy digestion of text. The *.disclaimer-msg* is features dramatic styling for capturing user attention without being overly stating it's presence.
- Controls – all buttons are consistent in sizing and focus states, maintaining a level of professionalism.

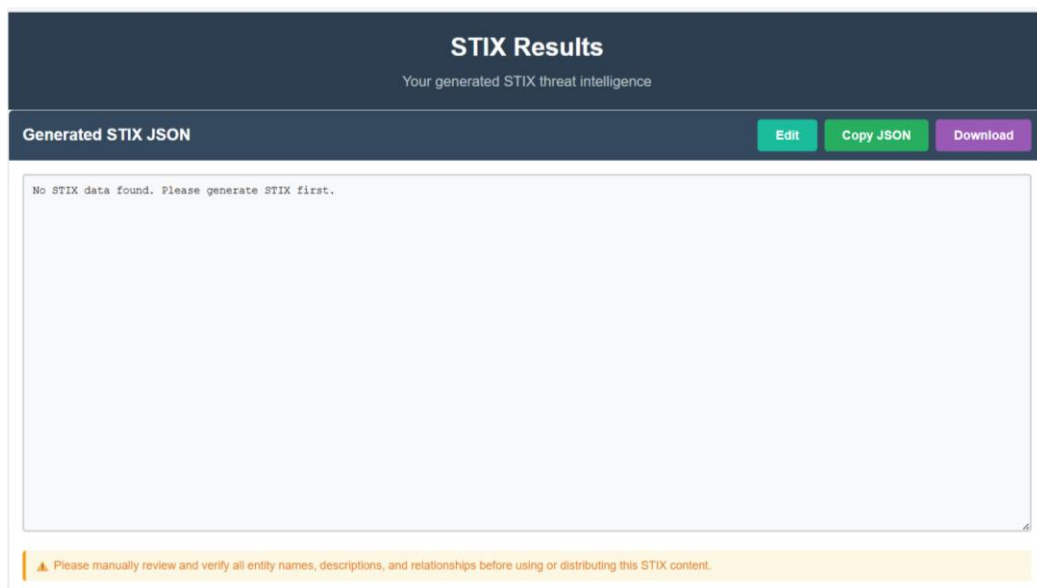


Figure 24. results page

4. Development lifecycle

This project evolved through three key milestones each with vastly different implementations, each iteration improving functionality and tightening compliance to standards. The overall approach is best described as iterative – incremental, aligned with agile methodology. With hindsight, the iterations are defined by a lean usable model deliverable, validation tests to assess functionality, identifying gaps in use and correctness, then repeating the process until arriving at a complete model. In each cycle verification and validation were the primary concerns, verification measures included input gating, defensive parsing and crucially, in later iterations, construction of STIX objects with the official library to avoid nonconformity with schema. Validation measures pinned focused on user centred flows, readable outputs, live suggestions and relationship modelling features.

In the earliest iterations, JSON bundles were manually assembled providing the intended functionality but exposed risk of schema drift. In later iterations, code pertaining to manual assembly was replaced with `sitx2` object builders, reducing the scope for error and enforcing schema compliance for SDOs, SCOs, SROs. As the model was refined, the frontend adopted a three-page view differing from its singular and double page predecessors, this change was prompted by a newfound understanding of user flow acquired over the course of development. This section will detail the lifecycle per iteration, with excerpts from codebases and explanations of foreign concepts, structured as requirement, design, implementation with example and testing.

4.1 Iteration 1 – NLP prototype

4.1.1 Meeting functional scope

The first iteration sets out the realistic but firm goal for accepting user natural language threat intelligence and transforming it into a JSON bundle compliant with STIX schema, whilst keeping output minimalistic but useable. The prototype focused on extraction of common Indicators of Compromise (IoCs) like file hashes, network addresses, domain names and email addresses, then assigning a descriptor like malware, attack pattern and threat actor – outlined in STIX standard. The application exposed a simple Flask API with routes for the single page UI and a POST endpoint that returned a formatted result. This naïve approach manifest itself in the early `STIXConverter` class and the `/api/convert` handler, where the server receives `{“text”: ...}`, applies the non-contextual pattern matching and returns an imitation of a JSON STIX object.

On the front end, a single HTML file features a text area for typing, a “Convert to STIX button” and a results section with NLP analysis and formatted JSON. This incomplete presentation with reduced complex logic and trace amounts of fetch logic made manual testing and validation simple. By entering threat intelligence, interacting with the UI, and inspecting output, the tool could be directly assessed for compliance.

4.1.2 Design

Iteration 1 inherited a minimalistic build, a single flask application file with local processing logic. This version leveraged a denomination of NLP called Natural Language Toolkit (NLTK) for sentence and word tokenisation, stop-word removal and stemming via the Porter algorithm. Tokenisation would split text into phrases and words, stop-word handled removing noise from text that didn’t aid context (e.g., “the”, “and”), and stemming normalised variants of words (i.e., attacking becomes attack) to enhance keyword extraction. The design also included compiled regular expressions for structured artifacts like *MD5* (*[a-fA-F0-9]{32}*), *SHA-256* (*[a-fA-F0-9]{64}*), IPv4 octets, domains, and emails. These patterns populate the internal “found” sets that later feed STIX object creation.

Where in the codebase. Tokenisation and stemming are in *process_text_nlp*, and the compiled IOC regex patterns are initialised in *STIXConverter.__init__*

4.1.3 Implementation

The extraction pipeline worked by three steps, IoC detection via regex, dictionary detection for malware and attack patterns aided by stemming, and lightweight heuristics for threat actor sightings (e.g., APT-style names, or generic actor words such as “group” or “hackers”). For example, a text like “We saw a phishing attempt contacting 192.168.10.1 and the domain evil.com” will trigger the malware list and the IP or domain regexes which are then rendered as a bundle where the IP or domain are indicators with STIX observable patterns such as *[ipv4-addr:value = '192.168.10.1']* and *[domain-name:value = 'evil.com']*. In Iteration 1 the bundle is assembled using created logic without using an external STIX helper library. It sets types, names, and IDs and packages them into a JSON object.

4.1.4 Testing

Verification focused on structural compliance with STIX schema and basic input hygiene. The server rejects insufficient input determined by short or empty value. The naïve */api/check* confirms that a becoming bundle is an object with “*type*”: “*bundle*” and an *objects* list. On frontend results are displayed with the number of NLP instances and extracted indicators, particularly helpful when validating against expected outcomes. Outputs should be consistent with that of the text used to measure compliance.

4.1.5 Results

The first iteration satisfied the outlined criteria of creating JSON format content from natural language user input, however, custom algorithms to achieve this meant many edge cases were not accounted for and thus overlooked. The system was at risk of schema drift, and missing context as a result of non-comprehensive regexes or dictionaries.

Due to the intense focus on SDO modelling, SRO logic was overlooked entirely. The absence of logic pertaining to this part of threat intelligence modelling weakened the overall value of the tool. These findings identified the areas requiring improvement and shaped the next iteration of the tool, the goal became tightening adherence to STIX schema for created entities and introducing SDO building logic for a more complete threat intelligence output.

STIX AUTOMATION TOOL



Figure 25. legacy UI implementation

4.2 Iteration 2 – Full redesign

4.2.1 New requirements

Iteration 2 redefined the scope of the system, with heavier emphasis on a complete workflow. Instead of adopting a one-page solution, this iteration introduced a:

- landing page for user orientation
- main page that aids authoring of threat intelligence with live suggestions
- results page for clear distinction of output for review featuring new export functionality

On the backend, the updated requirements forced a shift to an outsourced implementation where precision was the primary concern. For STIX standard compliance, stix2, an official package facilitating more accurate object construction compared to custom code replaced the existing logic. For broader extraction, custom logic was replaced with spaCy NER and IoC libraries and endpoints were introduced to incorporate live suggestions and relationship building functionality. This new ‘outsourced’ approach reduces the natural oversights of custom code, by delegating STIX serialisation to refined and tested classes, recall and precision is greatly improved.

4.2.2 Architecture

The rescope backend adopts spaCy for Named Entity Recognition for proper nouns like organisations, locations, individuals and groups iocextract for comprehensive IoC scraping, tldextract for precise domain derivation from URLs, RapidFuzz for fuzzy matching of a newly added suggestions feature and stix2 classes (e.g., Indicator, Malware, ThreatActor, Campaign, Tool, Identity, Vulnerability, ObservedData, Relationship, and File) are responsible for composing STIX bundles with full compliance. NER automatically assigns semantic labels to text spans (i.e., “Microsoft” becomes ORG), while fuzzy matching compares a user’s partially typed token against a dictionary and returns close candidates ranked by a similarity score. The server initialises spaCy’s English model and disables unused pipeline components to reduce latency, indicating a design focused on performance.

The frontend features a cleaner design split across three pages with attention to user experience. The landing page introduces the system and routes users into the flow. The main page provides the textarea input, a side panel with live suggestions, and a relationship drop-down for creating links such as in a non-technical fashioned environment. The results page shows the bundle in a read-only textarea with Copy, Download, and Edit toggles, plus a call to start a new analysis. The JavaScript modules (script.js and results.js) manage client-side events, call the back end, and pass the bundle between pages via sessionStorage.

4.2.3 Implementation

Entity extraction capabilities are enhanced greatly through use of complementary extractors, `iocextract` specifically built for spotting URLs, addresses, emails and hashes, `tlldextract` derives domains from URLs whilst existing regex finds CVEs and `spaCy` NER handles identities and locations for SDOs. This increases coverage significantly beyond the capabilities of regex alone, whilst simultaneously driving down errors proposed by edge cases (i.e., IP like domains caught under `tlldextract`).

Standards compliant STIX construction facilitated by `stix2` classes handles data with more authority. For example, URLs and domains become Indicators with proper STIX observable patterns (`[url:value = '...']`, `[domain-name:value = '...']`). File hashes are modelled as a File SCO (`File(hashes={...})`) wrapped in an ObservedData SDO, correctly separating an observation of a file from an analytic indicator about it, which previous logic could not differentiate. Threat actors, tools, malware families, vulnerabilities (CVEs), attack patterns, campaigns, and identities are each rendered via their canonical SDO types. The final bundle is created via `Bundle(objects=...)` and serialised by the library, reducing the risk of schema drift and guaranteeing required fields are present.

Relationship building addressed the issue of non-facilitation for SRO management. The client calls `/api/get-objects` to populate dropdowns with the set of extracted object names, enforces that source and target differ and that all fields are complete. When the user generates STIX, these pending relationships are posted with the text and incorporated by the server into the same bundle, producing Relationship objects linking the proper object IDs. This design places the user in control while keeping everything within a single, reproducible conversion transaction, adding to the threat intelligence modelling experience.

Live suggestions accounts for the inherent issue with natural user input, as threat intelligence is scribbled into the textbox it is possible for spelling mistakes to be made, recognising this problem and providing a solution enhances the user experience and solidifies a place in nature for this tool.

After remodelling the logic **user journey** was prioritised to ensure usability of the tool, with repeated testing an ideal modelling process with formed.

4.2.4 Testing and results

Correctness saw a huge improvement; by making the STIX constructors the only pathway to object creation rather than custom logic, verification was moved forward in the modelling pipeline as many invalid states cannot be created in the first place. With fixed results per instance less time was spent debugging structural errors and efforts were shifted to extraction quality, this differs from the previous iteration where `/api/check` only accounted for shape and not context.

Instead of reinventing the wheel, using preexisting technologies like `spaCy`'s pretrained NER or other imported tried and tested libraries resulted in increased recall with minimal operational overhead and custom code complexity, since there was no need to train any data like in the instance of machine learning. Disabling unused `spaCy` pipelines not relevant to this project meant latency was kept to an acceptable minimum, with a bigger application compared to its older version, this was particularly important.

Iteration 2, although being an improvement on iteration 1, still presented issues with naming consistency, contextual gaps and deeper user considerations.

Even with NER and other libraries for entity extraction, threat intelligence which arrived in many unformatted or ununified displays (i.e., dashes in names, ambiguous capitalisations) was occasionally overlooked, this weakened the extraction capabilities. A measure for normalising these instances was required to combat this issue.

The tool was also successful in populating objects with entities extracted from text, however, it failed to intelligently assign roles which are crucial to displaying threat intelligence correctly. For example, when

‘Microsoft’ is an identity, it cannot infer from text whether it should fall under threat-actor or victim. False categorisation of this identity could invalidate the output entirely and inspire less trust from users. Users can be observed to describe victimisation implicitly through language e.g., “target”, “attacks”, this realisation provides a solution for avoiding such instances.

Although live suggestions were available at this stage, the suggested words were not grounded in threat intelligence rich vocabulary. Without this the suggestions were limited to common words, often overlooking highly relevant terms, this meant suggestions were too sparse or too generic to be helpful.

4.3 Iteration 3 (Final) – Cached term integration and target inference

4.3.1 New requirements

Having achieved a standards compliant user driven conversion flow, iteration 3 sets out to improve precision and recall of entities and contextual automatic authoring of relationships between entities for better assignment in the background. To achieve this, the system must integrate a curated list of threat intelligence caches, each containing a specific type of information helpful to the construction logic. Caches are appended with inference logic within `app.py`, the expected benefits are better naming consistency and recognition of known actor names. These caches also provide predefined categories for otherwise ambiguous entities, for example, under the influence of *organisations.json* ‘Microsoft’ identified in text is never labelled as a threat-actor but rather an identity, and automatic inferencing connects actors to victims when the text implies as such through language. Live suggestions endpoint should also draw from the same cached files used during classification.

4.3.2 Architecture

The following datasets are read on startup: a large vocabulary cache (*ti_terms_cache.json*), a classification map that associates canonical terms to STIX types (*ti_classifications.json*), and curated lists of threat actors, organisations, and tools or software. An additional list of victim cue terms supports contextual victim detection. These files are loaded via a small helper that guards against missing or malformed resources, so the application remains usable even if the cache layer is incomplete. The loaded sets are normalised and organised into dictionaries for O(1) containment checks and quick canonical name retrieval.

4.3.3 Implementation

During extraction, the converter now cross-references tokens and n-grams against the vocabulary cache and then consults the classification map to decide which label the match should appear under (e.g., attack-pattern, malware, tool, campaign). This avoids misclassification of generic words and ensures that recognised strings become the correct SDO type. Where the input includes entities drawn from the actors, tools, or organisations lists, the system prefers those canonical names.

The converter also contains dedicated recognisers for threat actor naming conventions like “APT##”, These patterns are abundant in reports and when present should generate threat-actor SDOs even when other helpful indications to threat actors are not present like the word ‘group’ appended to it. Recognising them improves recall where adversaries are present in text, where proper names are not present, the system also accounts for instances where general terms are used providing a failsafe for sieved relevant data. When a campaign name is present, a cleaner function removes lead-ins (“observed”, “suspected”), strips trailing “campaign”, and formats the remainder as a title. When no explicit name exists but the text clearly describes a campaign or operation, an inference function synthesises a plausible label using topical keywords (e.g., “Phishing”, “Ransomware”) and a date in the text to generate an intelligible Campaign SDO name such as “Phishing Campaign 2025-08-13.” This raises the quality of the resulting bundle without fabricating new information, enriching the output threat intelligence and meeting expectations of users where information should clearly be actioned. Similar logic exists for other categories of SDOs to formulate a comprehensive and edge case inclusive system.

From prompt: “On 2025-08-11, APT29 conducted a spearphishing operation against ACME Corp staff in London. Messages lured employees to `https://login-microsoft-support[.]com` and a related domain, `login-microsoft-support[.]com`, which resolved intermittently during the activity window. Analysts referred to this as the suspected “Phantom Lure” campaign. Stolen credentials were used for VPN access shortly after initial compromise.” A created SDO featuring an inferred campaign name from text can be observed in results (fig. 26) whilst also demonstrating contextual automatic relationship building capabilities

```
{
  "created": "2025-08-15T00:09:14.493134Z",
  "id": "campaign--3f08c863-13f2-4d15-9973-8a1d5824817f",
  "modified": "2025-08-15T00:09:14.493134Z",
  "name": "Phishing Campaign 2025-08-11",
  "spec_version": "2.1",
  "type": "campaign"
},
{
  "created": "2025-08-15T00:09:14.493134Z",
  "id": "relationship--c9157d99-75b4-4caf-9128-4fbb14da3bce",
  "modified": "2025-08-15T00:09:14.493134Z",
  "relationship_type": "targets",
  "source_ref": "threat-actor--bf68f0e3-623a-45ce-adda-4477026dcecc",
  "spec_version": "2.1",
  "target_ref": "identity--919c3e16-2f32-4bae-88cd-ab0989547dc3",
  "type": "relationship"
}
```

Figure 26. inferred campaign name assignment

The live suggestion API calls on the same caches used for dynamic entity mapping, by doing so a context rich knowledge base can be feed into the system to provide more frequent and more relevant terms to users. Although the token system implementation remains unchanged, what has changed is the quality of data and the ability of RapidFuzz’s scorer to offer close and highly likely alternatives.

4.3.4 Testing and results

Canonical names reduce the chances of term duplication which may impair end results. For example, caching “Cobalt Strike” and “Lazarus Group” with the appended tool and group labels respectively ensures that user typos can be corrected by suggestion and that matches are serialised with consistent display names.

The inference logic is intentionally modest to keep programming overhead small, automatic links are made where possible and are results of proximity, strong matches with logic, and not guesswork. Human validation at the end encourages the user to oversee this limitation and adjust accordingly.

⚠ Please manually review and verify all entity names, descriptions, and relationships before using or distributing this STIX content.

Figure 27. disclaimer for tool use within results.html

4.4 Learning outcomes

Opting for existing technologies over custom logic

Using constructors that embed the rules of the chosen standard STIX is not only convenient from a programming perspective, but it was also a strategic shift that relocates verification into the model itself. By moving away from custom logic, tried and tested algorithms replace the workflow and account for a wider variety of instances.

Splitting the load for validation

High quality conversation is facilitated and partly orchestrated by the UI, by splitting the interface across three distinctive sections, each complete with guard rails (e.g., minimum requirements for text, read only outputs) the tool is turned into a guided experience enforcing quality at every stage.

Reducing complexity whilst maintaining functionality

Rather than train new models, a cached term approach improved recognition and naming consistency, and because the assets are plain JSON, they are transparent and easy to extend. The organisation, actor, and tool lists make the system's knowledge legible. This solution drastically reduces the computational needs of the system whilst simultaneously providing an adequate number of results to justify the trade-off.

Clear separation of concerns

Segmentation of the backend logic means that components can be swapped out for later iterations where a more suitable approach for tasks is identified. This modularity creates clean segmentation, so removing parts of logic will not destabilise the rest of the system.

5. Critical Analysis

This section appraises the project across its entire development lifecycle, addressing what worked, what didn't, and what would be done differently with the benefit of hindsight. It also elaborates on the software choices for development and outlines what should be done for future iterations, with enhancing threat intelligence modelling set as the primary goal.

5.1 Appraisal

5.1.1 What went right

Library incorporation

Moving from custom built JSON in tool prototypes to imported constructs featured in later iterations eliminated an entire case of structural errors. Fields necessary for compliance to STIX like IDs, timestamps and pattern types were enforced at inception rather than later (noncontextually) down the pipeline. This shift also meant hashes were modelled correctly as SCOs wrapped in ObservedData and not in Indicators – an oversight from custom logic. Compliance at creation meant verification was moved forward, reducing the possibility for error and thus increasing yield.

Frontend restructure

The updated user flow from landing through to results intentionally turned the UI into a validation surface. Implicit controls throughout the user journey like requirements for minimum word counts before generating STIX data or the requirement for toggling the edit button to manipulate outputs, provoke users to make deliberate choices for authoring. Enforcing attentiveness from users through these intentional hurdles encouraged high quality results.

Entity extraction without increased complexity

The project features a delicate balance between functionality and complexity. The tool, through relatively simple technologies, maintained a high level of extraction of entities without being at the cost of long computational times. This balance meant the tool was able to complete time sensitive tasks like live suggestions, provide exceptional outputs and reduce complexity of code, important for a task of this scope.

Cache driven precision

Curated lists of terms inclusive of actors, tools, organisations and various other threat intelligence related terms reduced spelling and format variance whilst improving extraction rates. By introduction of this feature the application was able to extract not only entities, but the context surrounding it, enriching the quality of data and subsequently, the final output.

Programming robustness

The backend application accounts for instance where logic important to functionality may be missing, whilst that would create problems with the quality of results, the system implements fail safes to ensure that the application does not crash and results are still produced. For example, if cache files are not available, the system then calls on a hardcoded set of terms to replace them. This adds resilience to the list of positive qualities this tool demonstrates.

5.1.2 What could be better

Inevitable outdateding of cache files

New threats emerge every day and manifest themselves across multiple categories such as threat actors, malwares, tools. The system calls on a large but not extensive list of threat intelligence data, however it does not account for all real-world observations or emerging threats, since this tool is aimed at modelling threat intelligence it can be assumed that future use cases will feature information not yet made available to the underlying applications. This oversight can make the tools functionality obsolete for new data; suggestions will become less helpful or frequent and cannibalization will become less reliable.

Missing context and conservative suggestions

Token focused hints combat latency, but where multiple related words or phrases are present, the current implementation may fail to recognise them. As a result, ambiguous parts of text which do not match any patterns covered by logic see a downturn in performance. This can manifest itself as missing live suggestions and incomplete extractions.

Manual relationship modelling

Although the system handles individual object modelling, users are still required to create their own SROs. Although presented as a drop-down menu for ease of use, this is not aligned with the intended functionality of the tool. Automation is the unique selling point of this tool; manual authoring of relationships should not be considered best practise. The current implementation streamlines complexity yet reveals substantial untapped potential for the modelling process.

Validation

Whilst constructors for preventing structural errors are an improvement on previous adaptations, validation is still limited. Like stix2, there are other official imports under the STIX brand which can be used for deeper post build validation for things like JSON schema checks or STIX linting. Introduction of such pipelines will help with robustness whilst maintaining a low complexity solution methodology.

Operational gaps

The application does not feature any production grade logic implementations; there are no built-in mechanism to gather analytics like suggestion acceptance rates or number of entities extracted per average length. For future iterations, this data may be useful for feedback loops and other optimisations.

Limited use case

The tool simply aims to tackle one problem – convert natural language user text into actionable JSON format threat intelligence. However, there is much more to threat intelligence modelling beyond the scope of this tool. STIX features an instance of this increased capability with a visualisation tool; introduction of this feature centralises STIX capabilities and solidifies purpose.

5.2 Future enhancements from hindsight data

The identified lapses from critical analysis provide actionable upgrades for future iterations. Each new improvement plans to increase the completeness of the tool whilst providing extra functionalities.

5.2.1 Auto updating knowledge base

The current implementation aims to model existing threats without consideration for emerging instances, for future implementation a new pipeline should be introduced to dynamically pull data from vetted sources like MITRE ATT&CK and directly import them into the existing cache, by this addition, the tool remains relevant, and its features remain highly functional. Updating schedule could be handled by Github Actions and jsonschema could be used to enforce formatting.

5.2.2 Intelligent automatic relationship modelling

The tool features a relationship drop down for ease of use in absence of an automated relationship constructor. For future, the extractor should also feature a new rule base relationship subfunction that intelligently deduces evidence from text with confidence. To prevent hallucinations, rules use verbs and light dependency patterns under window constraints. This could be handled by spaCy's Matcher or DependencyMatcher aided by relevant verb dictionary.

5.4.3 Advanced validation

Whilst current constructors prevent schema errors, more intricate issues are overlooked. stix2-validator is the official tool which checks whether STIX JSON content conforms for to specification and not just JSON schemas, and platforms rules that schemas can't express. With this implementation the STIX pattern syntax checker validates Indicator patterns and results from this are returned as errors or warnings.

5.4.4 STIX visualiser

Visualisation provides extra depth and easier digestion for threat intelligence information. For ease of integration an export-to-visualiser function at results.js could take output data and redirect to the URL with data already populated.

With later iterations and a more modern approach to creating UI, React (rooted in .js) facilitates the use of stix2vis, which assigns the official visualiser to a frontend component for an integrated solution.

5.3 Analysis of approach

The informal, natural progression of the tool was the right strategy for discovery. With the primary goal of turning user text into threat intelligence, each iteration addressed the risk most visible at the time. The first proved the tool could produce a workable STIX bundle, the second focused on correctness and user experience, and the third filled logical gaps with curated knowledge and light inference. This trial-and-error approach was anchored in real outputs rather than assumed results, which surfaced what users needed. The trade-off was spending time on custom logic for problems mature libraries and data sources already solve, adopting external assets sooner would have cut complexity between early stages, namely between iterations 1 and 2. That delay accumulated edge case rules, grew the codebase, and made milestones harder to hit.

A stronger upfront research effort would have surfaced higher value references, mature tooling, and design patterns early, cutting code production time and reducing task complexity by grounding choices in a clearer understanding from the start. Next time, the work would follow a more structured plan that translates research into a roadmap with explicit, timebound, and actionable goals with regular checkpoints to measure progress against those goals and deliberate adoption of proven resources where they fit. This shifts the emphasis from discovering foundations while building to building on well understood foundations, improving predictability, keeping the code smaller and clearer, and accelerating delivery without sacrificing trust or quality.

5.4 Unit testing

This section will evidence a comprehensive list of unit testing done on the application to assess its quality and robustness, it will outline a scenario with expected results and then a acquired result against specific areas within the application, labelled with task and ID.

id	Name	Testing what	Expected	Actual	How
t01	IP to Indicator	Extract IPv4 and create STIX Indicator	Indicator of IPv4 is in bundle	PASS	create_stix_bundle() emits indicator for detected IPs
t02	URL & Domain to Indicators	Extract URL and base domain into indicators	Domain name and URL indicators are created	PASS	tldextract + entity parsing feed indicator creation
t03	Email to Indicator	Extract email and create STIX Indicator	Indicator with [email-addr:value = 'alice@example.com']	PASS	Email regex/entity adds to indicators in bundle
t04	Hash to File SCO + Observed Data	Convert MD5/SHA* hash to File SCO + observed-data	File SCO with hash + observed-data referencing it	PASS	Hash detection builds SCO, observed-data emitted
t05	CVE to Vulnerability SDO	Detect CVE tokens and create vulnerability	Vulnerability object named CVE-YYYY-NNNN exists	PASS	extract_all_entities() collects CVEs; bundle adds SDO
t06	Threat actor code to Threat-Actor SDO	Detect APT/TA codes	Threat-actor SDO created (e.g., APT29)	PASS	Regex/list mapping to adds threat-actor
t07	Location (GPE) to Location SDO	NER location to STIX location	Location SDO created for 'London, UK'	PASS	spaCy NER GPE/LOC mapped to location SDO
t08	Campaign inference	Create campaign when attack/phish wording present	Campaign SDO added (e.g., 'Phishing Campaign <date>')	PASS	Keyword heuristics add campaign when present
t09	Summary counts in response	Return summary of bundle object counts	Response includes summary totals	PASS	/api/convert compiles and returns summary
t10	User-defined relationship	Create SRO between named SDOs	Relationship (e.g., 'uses') links sourcetotarget	PASS	id_map lookup and relationship construction succeed
t11	Auto 'targets' relationship	Auto TAtolidentity 'targets' when victim phrasing present	Targets relationship added when terms like 'targeted' appear	ERROR PRONE	Heuristic depends on victim terms and proximity; conservative linking skips borderline cases
t12	Prevent self-relation	Block same source/target or empty fields	Relationship not created; warning shown	PASS	Client-side validation stops invalid SROs
t13	Relationship list add/remove	CRUD UI for relationships	Added rows show 'A <type> B'; remove works	PASS	State array re-renders list; removeRelationship() updates
t14	Landing to Get Started redirects	Entry routing from landing	Clicking Get Started navigates to /main	PASS	Anchor href='/main' in landing.html

t15	Generate disabled <10 words	Input gating rule on main page	Generate stays disabled until 10 words	PASS	updateButtons() computes wordCount and disables button
t16	Generate after 10 words	Successful conversion flow	POST to /api/convert; redirect to /results; bundle shown	PASS	generateSTIX() posts, stores output, navigates
t17	Clear button behaviour	Reset editor state	Clears text, relationships, suggestions; buttons update	PASS	clearAll() empties state and calls updaters
t18	Live suggestions threshold	Suggestions only after 2+ chars	Default panel until 2 chars; then show up to 5 items	PASS	handleLiveSuggestions() early return <2; fetch thereafter
t19	Apply suggestion	Replace current word with selection	Word replaced; caret to end; panel resets	PASS	applySuggestion() rebuilds text and selection
t20	Results load from sessionStorage	Hydrate results view	Pretty JSON shown if present; else helpful message	PASS	results.js reads sessionStorage and formats JSON
t21	Results actions with no data	Disable edit/copy/download when no JSON present	Buttons disabled until valid JSON loaded	FAIL	results.html starts buttons enabled; JS doesn't disable on empty
t22	Results edit/copy/download with valid JSON	Operate on populated output	Edit toggles readonly; Copy updates clipboard; Download saves file	PASS	toggleEditable(), copyToClipboard(), downloadJSON() paths work
t23	Download invalid JSON blocked	Guard export on malformed text	Toast shown; no file downloaded	PASS	JSON.parse wrapped in try/catch; aborts on error
t24	Start New Analysis	Reset and return to main	Clears sessionStorage and navigates to /main	PASS	startNewAnalysis() removes key and redirects
t25	Large input scroll	Editor stability with big text	Textarea height fixed; vertical scrollbar appears	PASS	CSS fixed height; native scroll engages