Knowledge Graphs for Cybersecurity Reasoning

FINAL REPORT

Team 1
Benjamin Blakely - Client
Benjamin Blakely - Adviser
Brandon Richards - Frontend Development Lead
Micah Gwin - Python/ML Development
Alice Cheatum - Programmer
Nicklas Cahill - Tester/Programmer
Michael Watkins - Python/ML Development
Carter Kitelinger - Client Interaction

sdmay23-01@iastate.edu
https://sdmay23-01.sd.ece.iastate.edu/
Revised: 04/30/2023

Executive Summary

Development Standards & Practices Used

An essential development practice for our project is the proper use of version control such as Git in addition to testing standards. Everyone on the team uses feature branches and pull requests when writing code to ensure a clean main branch. These pull requests are to be reviewed by at least one other team member for feedback. In addition, pull requests can be easily reverted if there is an issue. Testing standards include minimum coverage of 75% for each module and 100% of tests should pass before code is committed. Specific specifications are also considered in this project, including:

- PEP8
- NIST
- CVE
- CVSS

Summary of Requirements

- The product should use no less than 3 sources for cybersecurity information.
- The product should crawl sources for new information at least once a day.
- The product should extract information using HTML parsing, OCR, and/or API.
- The product should perform Information Extraction by using a trained model.
- The product should produce a knowledge graph containing cybersecurity entities and relations.
- The product should persist this knowledge graph using the graph database Neo4j.
- Extensible by future developers by using interfaces and writing modular code.
- Code should have documentation on function definitions and up-to-date README.
- The crawl speed should be rated as to not disturb the resources of the sources.
- Only open-source libraries should be used in the product.
- Low-medium power GPU cluster for NER model training
- Responses to queries should support being displayed in graph or tree form.
- The user should be able to build queries using date, company, and vulnerability filters.
- Query results should be returned in less than 30 seconds.

Applicable Courses from Iowa State University Curriculum

- COMS 311

- CprE 310
- ENG 314
- COMS 309
- COMS 319
- CYBE 230
- CYBE 231

New Skills/Knowledge acquired that was not taught in courses

- Python
- Machine Learning
- Web Scraping
- Docker
- MongoDB
- TypeScript
- Neo4j
- Python Libraries: BeautifulSoup, scrapy, coverage, unittest, spaCy

Table of Contents

1 Team

1.1 Team Members

- Brandon Richards
- Micah Gwin
- Alice Cheatum
- Nicklas Cahill
- Michael Watkins
- Carter Kitelinger

1.2 Required Skill Sets for Your Project

- Python Programming Experience
- React
- Cyber Security Knowledge
- TypeScript
- Machine Learning Knowledge
- Docker
- MongoDB
- AWS Architecture

1.3 Skill Sets covered by the Team

- Python Programming Experience - All
- React – Brandon, Micah, Michael
- Cyber Security Knowledge – Alice, Nicklas, Carter, Michael
- TypeScript – Brandon, Michael
- Machine Learning Knowledge - Michael
- Docker – Brandon, Michael
- MongoDB – Michael
- AWS Architecture – Michael, Micah, Brandon

1.4 Project Management Style Adopted by the team

Our team has chosen agile as a project management style because our project goal of developing a knowledge graph will require many iterations. New knowledge and metrics that we acquire during sprints will help us better understand and plan for the next set of tasks we need to perform in the next sprint.

1.5 Initial Project Management Roles

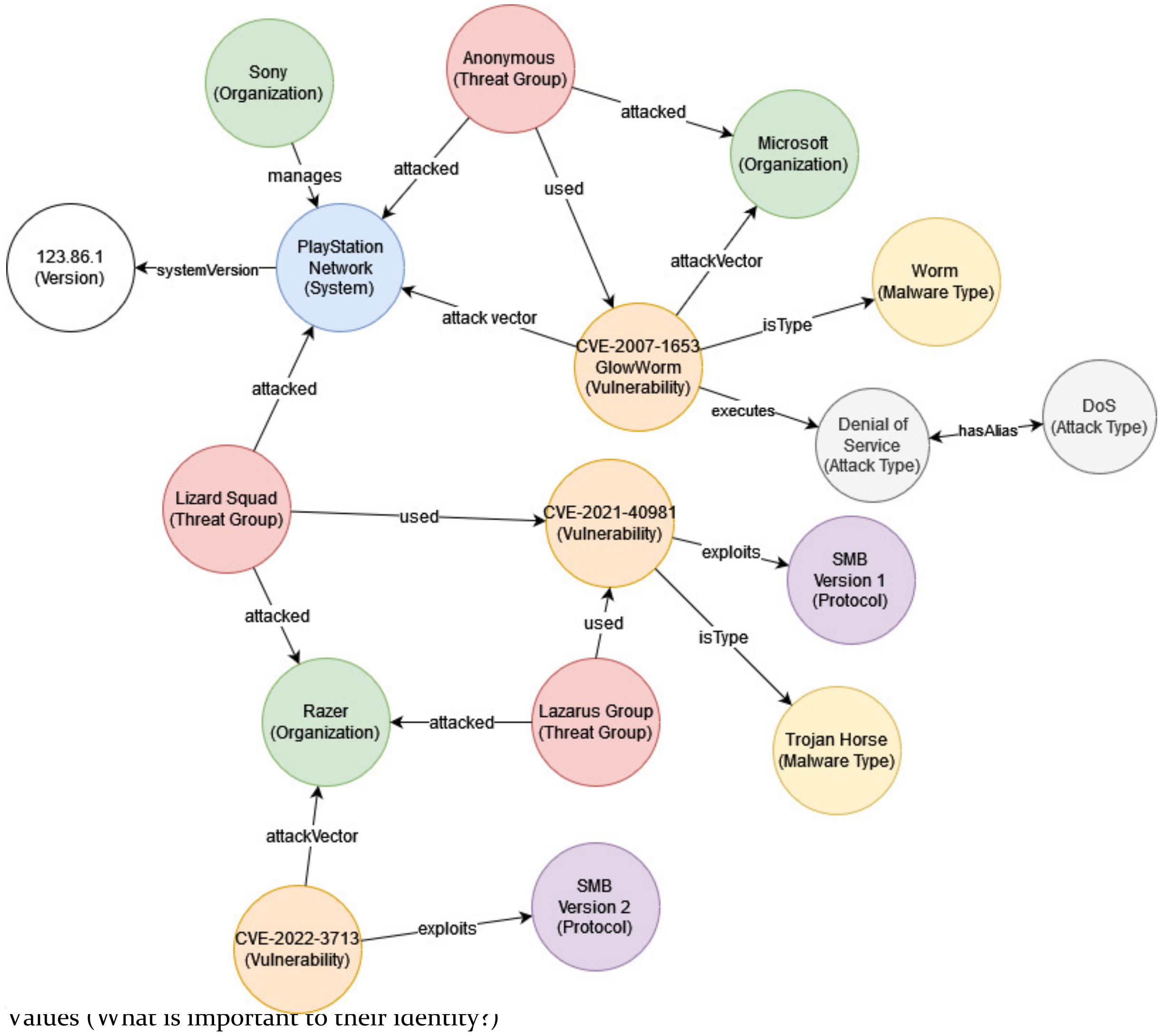
- Brandon Richards — Frontend Development Lead
- Micah Gwin — Python/ML Development
- Alice Cheatum — Programmer
- Nicklas Cahill — Tester/Programmer
- Michael Watkins — Python/ML Development

- Carter Kitelinger — Client Interaction

2 Introduction

2.1 Problem Statement

Cybersecurity threat reporting is currently spread out across multiple sources and written in a non-standardized format. Information is updated frequently, changing the landscape and requiring much effort to parse and read for relevant information. Cybersecurity researchers, Incident Responders, and System Administrators need to be able to efficiently query information about a specific software, malware, threat, etc., as well as new and emerging ones. Generating a Cybersecurity Knowledge Graph (CSKG) that contains relevant datapoints will allow for efficient information storage and querying capability.



Values (What is important to their identity?)

- Anonymity
- Privacy
- Informed

2.2.1.2 Empathy Map

Who? **Cybersecurity Researcher**

What / need to do?

- Be knowledgeful of relevant current threats.
- Understand context and implication of threats

See?

- News articles / research papers of new threats
- Attacks against enterprise and personal computer systems.

Say?

- I wish there was a quicker and easier way to find this stuff!

Hear?

- Other researchers talking about cybersecurity.
- Queries about how a cybersecurity threat affects a specific entity (company, university, software, etc.)

Do?

- Look at scattered reporting of cybersecurity threat.
- Parsing for relevance.

Think?

- I hate having to parse through many publications to find relevant information
- I hate having to maintain a list of reliable sources

Feel?

- Determined
- Curious
- Frustrated
- Annoyed
- Overwhelmed

Need Statement:

A Cybersecurity Researcher needs a way to parse relevant information quickly and efficiently because the landscape changes rapidly and sources are spread out and contain irrelevant details.

Benefit:

Researchers would see a reduction in time spent searching for new and related information, leading to better context and comprehension.

2.2.2 Incident Responder

2.2.2.1 Persona

Demographics

- College Grad
- Various Certificates

Hobbies and interests

- Penetration testing
- Keeping networks secure
- Cybersecurity

Motivations (Who do they want to be? What do they want to do? How do they want to feel?)

- Protecting business operations
- Protecting client information

Personality and emotions

- Investigative
- Curious
- Defensive
- Eye for small details

Values (What is important to their identity?)

- Intelligence
- Competence
- Integrity

2.2.2.2 Empathy Map

Who? **Incident Responder**

What / need to do?

- Respond to network intrusions, access policy violations, cybersecurity threats
- Defend systems owned by their employers from attacks in the future

See?

- Current threats or intrusions to the company/business they are working for

Say?

- "I wish I knew of a quick and easy way to find information about this new vulnerability!"

Hear?

- Is our infrastructure safe?
- We've had a network intrusion; you need to fix this.

Do?

- Investigate and patch exploited systems

Think?

- Which software or hardware flaw is responsible for this intrusion?
- Who attacked us?

Feel?

- Attacked
- Defensive
- Rushed
- Panicked

Need Statement:

An Incident Responder needs a way to find vulnerabilities quickly because investigating and patching cybersecurity threats requires up-to-date information on a time crunch.

Benefit:

Quicker information gathering and analysis results in a faster response time to threats and stronger defenses in place for next time.

2.2.3 Sys Admin

2.2.3.1 Persona

Demographics

- At least Highschool Grad
- Certificates

Hobbies and interests

- Software or hardware systems
- Networks
- Servers
- Maybe a mild interest in cybersecurity

Motivations (Who do they want to be? What do they want to do? How do they want to feel?)

- Maintain the systems they are responsible for, focusing on uptime and usability.

Personality and emotions

- Meticulous
- Overworked
- Problem solver

Values (What is important to their identity?)

- Efficiency
- Network/server uptime
- Accessibility
- Supporting end-users

2.2.3.2 Empathy Map

Who? **Sys Admin**

What / need to do?

- Keep the systems they are responsible for secure
- Know which threats are most relevant
- Balance security with usability of the systems

See?

- News articles about new vulnerabilities, exploits, and attacks

Say?

- Why isn't this working?
- What new vulnerabilities are there for the software we run?

Hear?

- Why isn't this working?
- Wasn't this supposed to be secure?
- I thought you maintained this?

Do?

- Maintain hardware and software on many systems

Think?

- I dislike having to keep up with the constant cybersecurity knowledge while still needing to maintain systems

Feel?

- Overwhelmed

- Unfamiliar

Need Statement:

A Sys Admin needs a way to learn about current cybersecurity threats without in-depth knowledge because their systems need to be secure but they also have other things to focus and work on.

Benefit:

Can save time understanding cybersecurity problems, allowing for communication with others, and making more time for administrating systems.

2.3 Requirements & Constraints

Functional Requirements

- The product should use no less than 3 sources for cybersecurity information.
- The product should crawl sources for new information at least once a day.
- The product should extract information using HTML parsing, OCR, and/or API.
- The product should perform Information Extraction by using a trained model.
- The product should produce a knowledge graph containing cybersecurity entities and relations.
- The product should persist this knowledge graph using the graph database Neo4j.

Non-Functional Requirements

- Extensible by future developers by using interfaces and writing modular code.
- Code should have documentation on function definitions and up-to-date README.
- The crawl speed should be rated as to not disturb the resources of the sources.
- Only open-source libraries should be used in the product.

Resource Requirements

- Low-medium power GPU cluster for NER model training

Aesthetic Requirements

- Responses to queries should support being displayed in graph or tree form.

User Experiential Requirements

- The user should be able to build queries using date, company, and vulnerability filters.
- Query results should be returned in less than 30 seconds.

2.4 Engineering Standards

- PEP8

    - PEP 8 is the official style guide for Python code. Our team's code will follow this style guide to ensure maximum readability and avoid developer issues due to different coding styles in the same project.
- NIST
    - NIST has many definitions on various cybersecurity topics that we will utilize in our project. Our project will provide quick and easy information that has ties to NIST Cybersecurity Framework (CSF) and NIST Risk Management Framework (RMF), thus these standards will be incorporated into the end product.
- CVE
    - Common Vulnerabilities and Exposures is a standard for computer security flaws our project will need to follow to obtain, process, and serve our cybersecurity information. Using CVE will keep vulnerabilities tied to their initial reports, which include a description of the flaw, making it easier for users to follow a chain of information from our knowledge graph.
- CVSS
    - The Common Vulnerability Scoring System will be used in our project due to our interaction with vulnerabilities. This framework has three metrics that are used to rate the severity of vulnerabilities our knowledge graph will contain. We will interpret and display data in a CVSS format to remain consistent with other sources of information and improve comprehension by our users.

3 Project Plan

3.1 Project Management/Tracking Procedures

Our team has chosen agile as a project management style because our project goal of developing a knowledge graph will require many iterations. New knowledge and metrics that we acquire during sprints will help us better understand and plan for the next set of tasks we need to perform in the next sprint.

Our team is going to use Git as a VCS and GitHub to host our repository. We will also be using JIRA as a ticketing system to better track tasks, subtasks, sprints, backlog items, and responsibility for each task.

3.2 Task Decomposition

1. Determine list of sources to obtain news articles and blog posts.
    a. Inspect sources for legitimacy and reputation
    b. Record URLs of main page and/or specific pages of interest (e.g., Malware blog)
2. Develop scraper to obtain news articles, blog posts, etc., using Scrapy
    a. Select a programming language
    b. Select libraries to perform downloads and parsing of HTML
    c. Create file specification for storing list of sources

    d. Write code for scraper
    e. Write testing code for scraper
    f. Output artifacts for later use by NER model
    g. Containerize with Docker
3. Develop one or more methods to clean up articles (may vary depending on type of article)
    a. Research existing methods for cleaning up irrelevant information
    b. (Potentially) Manually annotate relevant vs. irrelevant data in documents
    c. Verify on test cases that relevant information isn't being destroyed
4. Extract relevant entities (vulns, companies, software, exploits, etc.) and the relationships between them
    a. Research existing annotation techniques and cybersecurity-specific NER models
    b. Determine if we need to train custom NER model
    c. (Potentially) Train NER model:
        i. Manually annotate set of documents from selected sources
        ii. Perform supervised machine learning to train NER model
    d. Generate entities and relationships from cleaned-up source information
5. Use extracted entities and relationships to generate knowledge graph
    a. Collect output from Information Extraction
    b. Insert into graph database
6. Run pipeline created in steps 2-5 periodically and continuously on new articles
    a. Create job to run at interval
    b. Determine if new articles of interest have been posted
    c. Run new articles through pipeline
7. Develop a web interface to run queries on the graph
    a. Design
        i. Develop prototype
        ii. Receive feedback from client
    b. Develop
        i. Select framework to make website
        ii. Create API to query knowledge graph
        iii. Implement designs from prototype
        iv. Implement filters to query the graph
        v. (Stretch goal) Use natural language to query the graph

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

1. Sources
    1.1. 3 or more sources have been selected to scrape for information
2. Scraper
    2.1. Scraper can download and parse input sources.
    2.2. Runs inside Docker container.
    2.3. Identifies more recent unprocessed articles with 100% accuracy.
3. Article cleanup
    3.1. Articles achieve removing unnecessary information with 25% accuracy.
    3.2. Articles achieve removing unnecessary information with 50% accuracy.
4. Extract entities and relationships

    4.1. Software can identify subjects (companies, operating systems, vulnerability, etc.) with 75% accuracy.
    4.2. Software can identify relationships (vulnerability works on this OS and application run by this company) with 50% accuracy.
5. Generate knowledge graph
    5.1. Contains more than 15 entities including companies, operating systems, applications, malware, etc.).
    5.2. Nodes have properly labeled edges with 75% accuracy.
6. Pipeline periodic and continuous running
    6.1. The job runs on the specific interval 100% of the time.
7. Web Interface
    7.1. Prototype delivered to client with 80% satisfaction (satisfaction to be quantified with rating survey).
    7.2. API created to query 100% of knowledge graph entities and relationships.
    7.3. Filtering by Company, Application, OS, Vulnerability, or Malware can be performed.
    7.4. (Stretch) Natural Language query can serve intended results 50% of the time.

3.4 Risks And Risk Management/Mitigation

Description	**Likelihood**	**Consequences**	**Risk**	**Mitigation**
Training uses too many resources	Unlikely	Major	High	Optimize code, set resource usage limits, allocate more GPUs for cluster
Model is trained incorrectly	Moderate	Moderate	High	Start early, involve Michael in most decisions due to experience, Research
Source is too difficult to perform extraction	Moderate	Moderate	High	Pre-scout source's format, plan extraction logic ahead
Resources too high to display query	Moderate	Moderate	High	Limit number of displayed responses
Sources go down or block our traffic because of too	Likely	Major	Extreme	Strict rate limits

much scraping too fast				

4 Design

4.1 Design Context

4.1.1 Broader Context

Our project is primary focused on two communities: security researchers in academia and information security roles in industry. Although these communities will be directly affected by our project, the entire world will be indirectly affected as well. The increased efficiency of gathering cybersecurity knowledge will allow our target communities to better do their job and, increasing safety, security, privacy, and integrity in all software applications. As societies around the world grow more dependent on technology, our goal is to make the software they interact with safer.

Area	**Considerations**
Public health, safety, and welfare	All users of a technology companies' products are indirectly affected by our project due to the utility we provide information security staff. Helping these roles increases the public safety and welfare in their interaction with technology. It could also harm job opportunities of these positions, as an effective product would require less staff to research and fix problems. In the same way, researchers' use of our product will also improve the general populations' interaction with technology utilizing their discoveries.
Global, cultural, and social	Our project accurately reflects the values of our target cultural groups including security researchers in academia and information security staff in industry. Using extracted information to build a knowledge graph that can be queried is in line with practices to streamline cybersecurity information gathering.
Environmental	The environmental effects of this project are indirect. The software will be executed and hosted on servers that use a lot of electricity of unknown origin (renewable vs. nonrenewable). There is a potential impact of training ML models with GPU clusters in terms of energy usage, although our project would have to scale magnitudes larger for this to become a reasonable concern.

Economic	This project being successful will have an impact of increased productivity by information security roles in industry by getting them access to recent, condensed, and relevant information quicker. One pending consideration is if the project proves extremely useful what obligation we have to make it available to good actors in terms of cost.

4.1.2 Prior Work/Solutions

There have been research papers on the idea of using Knowledge Graphs to store information in the Cyber Security domain. One example is “TINKER: A framework for Open source Cyberthreat Intelligence”. This research paper delves into creating a knowledge graph that is used primarily to “infer threat information from the [cybersecurity] text corpus”. This differs from our project in that it attempts to strictly capture malware information from CTIs (Cyber Threat Intelligence). Our project is aimed more for researchers and industry professionals to be able to query a knowledge graph of many entity-types (companies, vulnerably, malware) that is consistently updating with new information automatically.

Some of the pros and cons of our target solution would be:

1. Pro: Web Interface with query input and data visualization
2. Pro: Updating periodically (e.g., every hour) with new cybersecurity information
3. Con: Information sourced from reputable cybersecurity blogs instead of CTIs.

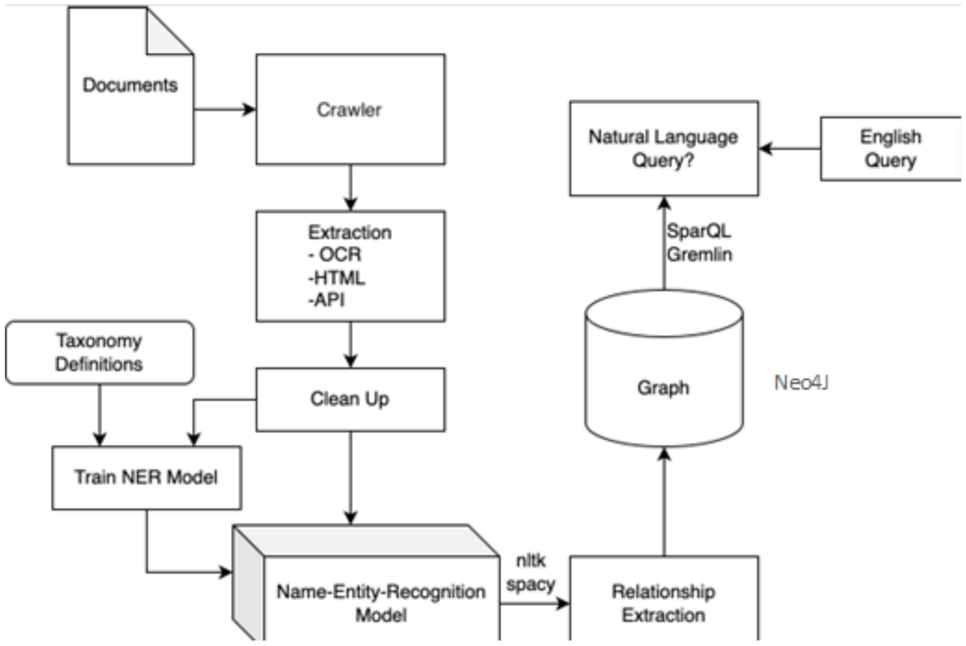
Our project is not following previous work of any Senior Design project.

4.2 Design Exploration

4.3 Proposed Design

4.3.1 Overview

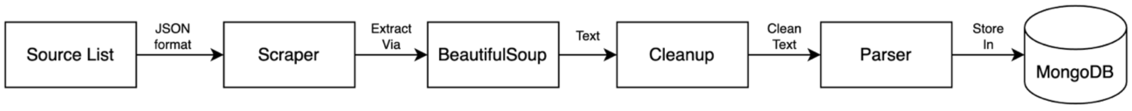
Our current design includes multiple software components, each with a role to play to achieve our overall goal. One component is responsible for collecting data from external sources, another for extracting subjects and relationships between them from that data, and another for saving this information in a graph. This design works as a pipeline to start with raw text information and end up with a graph that can be queried for information. One last component is a frontend that users can use to perform the queries.



4.3.2 Detailed Design and Visual(s)

4.3.2.1 Scraper

The scraper is a Python module responsible for inputting a list of sources and outputting text files. The source list input should be in JSON format and be an array of objects, each with a "url" property that points to the URL of the main feed. Each output text file should be the text from an article not yet scraped from the inputted source list. Scraping should be done using the Python library Scrapy, and extraction of the scraped text should be done using the Python library BeautifulSoup. The next subcomponent of the scraper is the "Cleanup". The input will be the extracted text and the output will be text without irrelevant information. This will require little or much effort depending on the source and thus should be examined on a source-by-source basis. The last subcomponent of the scraper is a database to store information including the URL fetched, the article text, and a timestamp of when it was collected.



4.3.2.2 Parser

The parser is a Python module that takes input from the scraper (text files) and performs Named-Entity-Recognition (NER) and Relationship Extraction (RE). These two jobs allow for the creation of a graph structure with Named Entities as the vertices and relationships as the edges. The output of this component are entities and relationships for the knowledge graph.

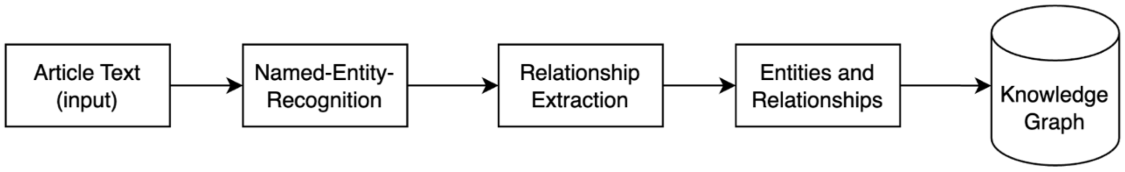
The NER subcomponent will extract the following types of entities from text:

- Organization

- Vulnerability - CVEs or named/known exploits
- Threat Group - Known malicious groups
- Malware Type - Virus', Trojans, Ransomware, etc.
- System - OS's, Hardware, Software
- Protocol, with the version if applicable

The RE subcomponent will extract the following relationships between entities:

- manages
    - Organization -> System
- attackVector
    - Vulnerability -> System
- used
    - System, Protocol, Filetype, Filename -> Organization, Threat Group, System
    - Port -> Protocol, System
    - Vulnerability, URL, IP -> Threat Group, Malware Name
- attacked
    - Threat Group -> Organization
- exploits
    - Vulnerability -> Protocol
- isType
    - Vulnerability -> Malware Type



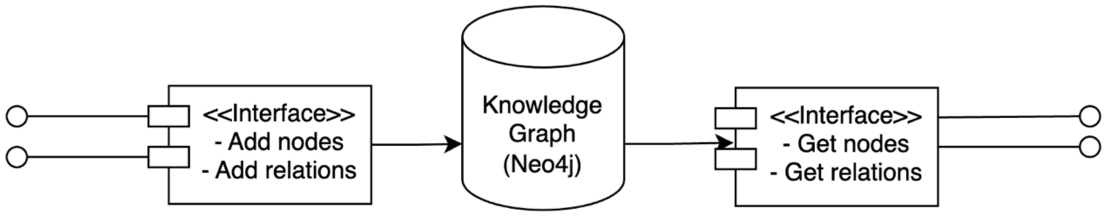
The parser currently has a couple of defined interfaces. They are as follows:

- AbstractInput
    - Description: Reads input from implementation-dependent source (MongoDB, file, hardcoded, etc.)
    - Methods:
        - get_inputs() -> list[str]
- AbstractParser
    - Description: Performs IE (NER + RE) on input and outputs entities and relationships
    - Methods:
        - parse(input: str) -> ParserResult
- AbstractOutput
    - Description: Outputs entities and relationships in an implementation-dependent way (file, database, etc.)
    - Methods:
        - output(result: ParserResult) -> bool
- ParserResult
    - entities: list[str]

        - relationships: list[ParserRelationship]
- ParserRelationship
    - from: str
    - to: str
    - type: str

4.3.2.3 Knowledge Graph

The Knowledge Graph component will accept entities and relationships as input and store these into a graph database. Our current choice for the database is Neo4j as it offers good performance, no cost, and is graph based. The entities will be stored as vertices in the graph (called Nodes in Neo4j) and relationships between entities will be stored as edges in the graph (called "relationships" in Neo4j). This should be running in a Docker container but have an exposed API that the Parser is able to use to insert these entities and relationships. It should also expose an API to perform queries on the graph, such as getting all nodes related to a node by some relationship.



4.3.2.4 Frontend

The frontend of our project will be a React web application written in TypeScript. The web app will accept user input specifying what information should be queried and will perform an API request on the Knowledge Graph. An example could be a cybersecurity researching searching for "Samsung" and getting back all or a limited set of nodes connected to it, such as recent attacks launched against them. It should then display the result in a graph or tree format.

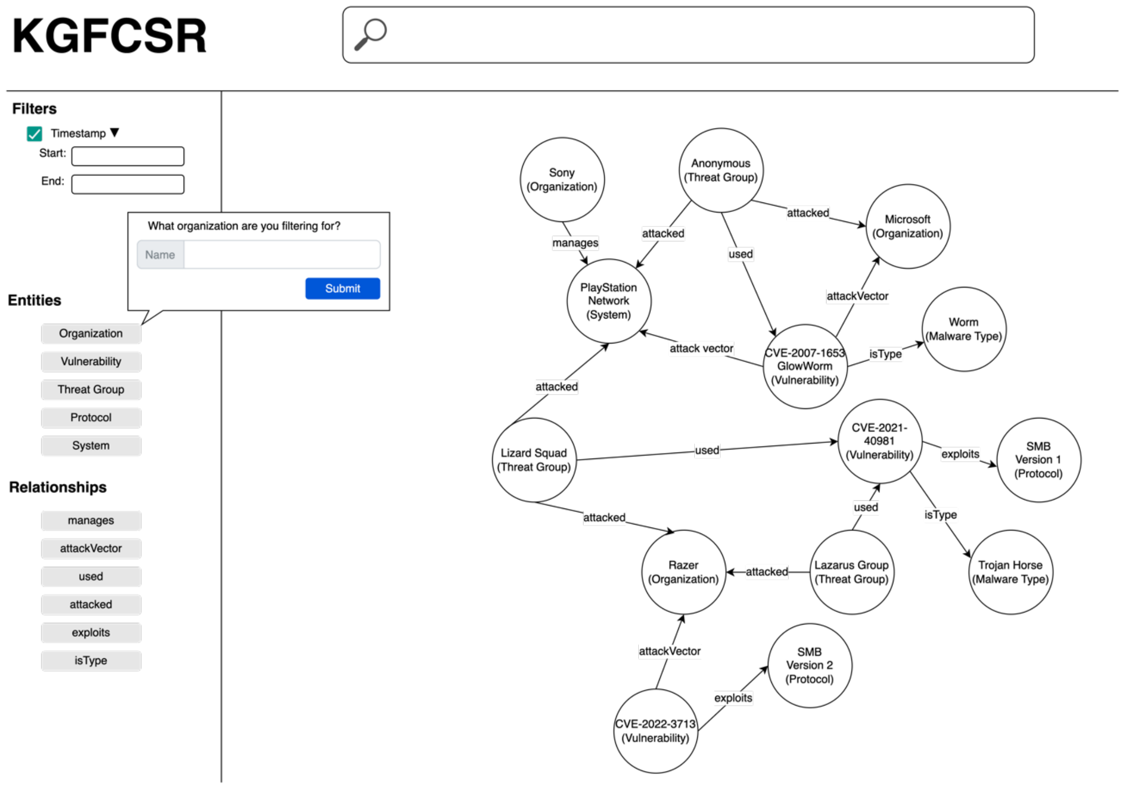


Figure: Frontend Mockup

4.3.3 Functionality

Our design is intended to operate by the user visiting a web application. The user will enter in a query, such as a company, vulnerability, and/or timeframe they are searching for, and the Knowledge Graph will be queried with those parameters. The user will be able to navigate around the Knowledge Graph by dragging the mouse in different directions to explore the connections between nodes. The user may also choose to have their results displayed in a Tree View rather than a Graph View.

If the user enters in a query for which there are no results, they will be prompted that no results exist. In the case of an error in the query, the user will be prompted that their query was completed unsuccessfully along with any additional error information from the server.

Periodically the pipeline will run again, scraping new articles, parsing them, and inserted the extracted information into the knowledge graph. The interval is currently defined as 1 hour, although this is subject to change.

4.3.4 Areas of Concern and Development

The current design completely satisfies the client requirements and moderately meets the expected needs of our users. The area of most concern will be the development of the Named-Entity-Recognition model and performing Relation Extraction due mostly to many unknowns we have yet to encounter. Performing NER and RE is on track to be the most complex portion of the project. The immediate plan for developing the solution to this component is beginning testing whether we need to train our own models to perform these steps or if we may take advantage of existing technologies such as CyNER. If we can use a previously trained model for the NER step, this would drastically decrease the level of effort needed for a successful Parser component. We currently have no questions, as our client and faculty adviser has graciously provided us with scientific papers going into details about different attempts at Cybersecurity Knowledge Graphs and some Information Extraction on a text corpus.

4.4 Technology Considerations

Scraping – scrapy

For scraping we chose the scrapy library. It is one of the most popular Python web-crawling frameworks. It features a wide variety of inbuilt tools to help us collect the data we need. Ultimately, we chose it because of the rich community and how widespread the framework is.

One of the weaknesses of scrapy is how basic its parsing tools are. Scrapy is great for collecting data from websites, but not the best at parsing that data. A solution, which we are implementing is to use another framework to parse the data. We chose BeautifulSoup4 due to its rich feature set, some team member's prior knowledge and its widespread usage.

NER – spacy

Spacy will be a great library to assist with NER and/or RE due to the good support available and that some team members and our advisor have some background in this framework. There exist other options such as the NLTK, and cloud offerings. We did not want to use a proprietary framework like the Google Cloud Natural Language API to keep our project open source and free.

NLTK is more academic focused that spacy. It is meant to be a toolbox of machine learning tools for academic use, while spacy is more oriented towards developers. Spacy has a richer set of tools to help us accomplish our goal faster and more efficiently. NLTK focuses just on strings, while Spacy focuses on objects. Most of our data is objects, so spacy was the better fit.

Database – SQL (column based) vs Neo4j (graph based)

The decision to use a graph-based database was a very easy decision. We are building a graph, so a graph-based database fits our data type. Graph-based databases use more memory and are harder to do text queries on, but store trees and graph data more efficiently than a column-based database.

Furthermore, a graph-based database allows storing arbitrary data. A column-base database requires data to be in a very rigid format – data must fit into the specified columns. A lot of our data is wildly different, with types such as vulnerabilities, software packages and malware. This variance makes storing data in columns difficult and unscalable.

Web technologies & NLP stretch goal

Our project will make use of TypeScript and the React library. We came to this decision over other web development frameworks because of a large ecosystem of packages, previous experience of team members, and the easy-of-use of performing API calls and displaying the data in an aesthetic way. Other considerations were made such as Angular, Vue, or simply HTML/CSS/JS. Ultimately because of the previously described reasons, React won as the library of choice.

4.5 Design Analysis

Scraper:

- Implemented basic scraping and parsing functionality, scraper reads input of sources in a JSON document and outputs the parsed HTML document for each source.
- Integrated the BeautifulSoup4 parses in with the Scrapy spider. Now instead of raw HTML data, it is being cleaned up to a more readable format which will make storing in a database or running language processing on the data much easier.

Docker Container and GitHub:

- Constructed new folder setup in GitHub that allows each folder to be a Docker container.
- Used Docker Compose to organize and build each container.
- The scraper and MongoDB database that was constructed to store article information will each be a separate container.

Article Tagging:

- Began using the NER Annotator tool available online to create a model which can be used by the spacy tool to process the article information.

So far during the implementation of the items above, the proposed design has functioned well in organizing and linking the different components together. The scraper and the parser are currently being worked on however the other sections of the proposed design have not been started yet so an analysis of how they are working is not possible. For future design, the article tagging will continue after more information is scraped and stored in the database. Then the created model from NER tagging will be used to parse the article data and identify information to build the knowledge graph

5 Testing

Our project will make use of many types of testing to test the various software components and subcomponents. For each of our components, our goal is to have at least 75% code coverage and 100% of tests pass. Each component is responsible for containing unit, integration, and system tests that verify its behavior internally, while additional integration and system tests will be outside of these components to test their interaction.

When changes are made in one component, it will be required to run all tests in that component in addition to all integration and system tests. New tests should be added to verify the behavior of newly added logic. Our main instruments for testing will be the Python testing framework unittest for our components written in Python and the testing framework jest for our website component written in TypeScript. This will work to run unit tests on individual components and integration and system tests across components.

The team's overall testing philosophy is to test early, test often, and use it as a tool. Testing can be a wonderful tool in a Test-Driven Development (TDD) framework because expected inputs and outputs can be specified first, then the tests' results indicate to the developer whether they've implemented the desired functionality.

5.1 Unit Testing

Scraper

The scraper tests and code coverage should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `scraper`
3. Run `make coverage` command
4. Verify no failures in tests
    a. If all tests pass, the text "OK" will show at the bottom of the output.
    b. If one or more tests fail, the bottom of the output will read "FAILED" with the number of failures in parenthesis (e.g., "FAILED (failures=1)")
5. Open generated coverage file `htmlcov/index.html`
6. Ensure test coverage is greater than or equal to 75%.

Parser

The parser tests and code coverage should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `parser`
3. Run `make coverage` command
4. Verify no failures in tests
    a. If all tests pass, the text "OK" will show at the bottom of the output.
    b. If one or more tests fail, the bottom of the output will read "FAILED" with the number of failures in parenthesis (e.g., "FAILED (failures=1)")
5. Open generated coverage file `htmlcov/index.html`
6. Ensure test coverage is greater than or equal to 75%.

Frontend

The tests and code coverage for the website component should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `frontend`
3. Run `yarn test` command
4. Verify no test failures

Pipeline Tests

During the "Build and Test Scraper" step in the pipeline, before the final zip is made and uploaded to the lambda first the step runs the "make test" and "make clean" commands to verify the scraper lambda code is working before it gets deployed. This requires no user intervention besides creating the tests exactly the same as you would for local ones and pushing them to GitHub for the pipeline to process. The same process occurs for the frontend, where unit and UI tests are run by the pipeline. The pipeline will fail if any tests fail.

5.2 Interface Testing

Our interfaces include the source list for the scraper, the entity-relationship output from the parser, and the graph API queried by the frontend to be rendered. To test these interfaces, we have tests in place that send data through the interface and verify its integrity. In addition, the parser entity-relationship output can be tested by inputting mock article data and ensuring the parser output is as expected. For most of our existing and future tests we use the Python testing library such as unit test or mocking tools in a test class that asserts the scraper and parser are functioning properly. The graph API is provided by Neo4j and thus isn't tested in our project, although any bindings that we write for this API will be.

5.3 Integration Testing

The critical integration paths in our project begin with the human-developed source list being handled correctly by the scraper. The next is the scraper properly storing article information in the MongoDB database. Then after this is completed, the parser retrieves the MongoDB data and constructs the list of entities and relationships. Lastly this entity-relationship data is stored in the Neo4J graph database and then queried by the frontend for rendering. All the above paths are essential to the project operating correctly and can be tested by using Docker Compose to build and run all the containers on a local or cloud server, then use a script or library to verify the container are communicating properly with data expected by that test case.

5.4 System Testing

By using source lists with inputs for which the expected outputs are known we can test that the output of our system falls within our defined parameters. Our first step to test output we will use *portions* of documents we annotated manually to ensure the machine learning program is outputting what we expect it to and compare it to the accuracy requirements of the annotation. As the machine learning portion gets fed more documents and learns how to annotate with more accuracy, we will begin to give it larger portions of a document until it can annotate a full document within the accuracy percentage requirement we defined. Once this point has been reached, we will begin to give it full documents until the machine learning has been refined enough to where it is no longer required for us to test each output for accuracy. Our system testing will use all the tools mentioned in the unit, interface, and integration test sections to test the whole system, as each section needs to perform to our standards for the whole program to perform well. If one piece is not working up to standards, the output may contain unexpected behavior.

5.5 Regression Testing

Our strategy to implement Regression Testing is supported by two rules: Have a separate, clean, working copy of the code on our main branch and run unit, integration, and system tests on every pull request to this branch. The pull request is not able to be merged until all tests are passing. This will ensure existing functionality has not been broken, plus having the main branch be a clean working copy allows us to rollback changes that break existing functionality. Critical features that our project needs to ensure do not break are the scraper fetching sources and the parser processing these sources and outputting expected entities and relationships.

5.6 Acceptance Testing

Acceptance testing of our functional requirements will be performed by running unit, integration, and system tests to provide verifiable evidence that our test cases are passing. Most of our project's non-functional requirements can be summed up as a satisfactory level of code quality, libraries being open-source and a license allowing free usage, and documentation of code. These non-functional requirements can be verified by reviewers of pull requests. For example, if a new library is added as a dependency, the reviewer should be double checking the license allows us to use it. Our client will be involved in acceptance testing by being made available the number of passed tests and code coverage, ensuring we've met the expectations set in the requirements and metrics/evaluation criteria. They will also be delivered our iterative beta builds after each sprint to evaluate our progress and correctness in our design implementation.

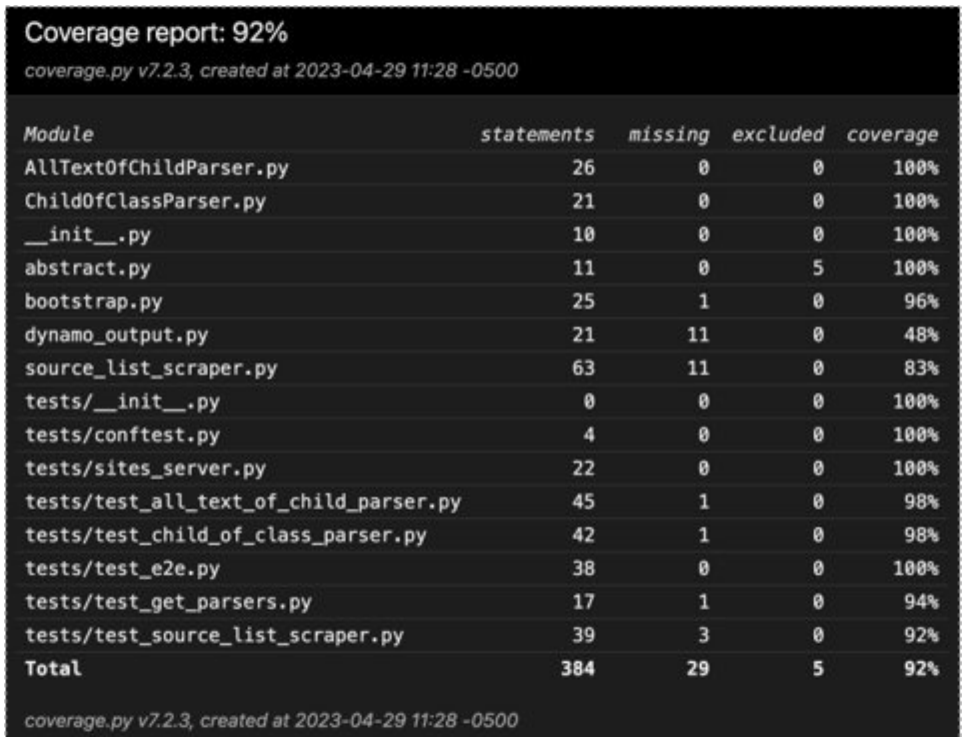
5.7 Security Testing

Security Testing will consist of testing that malicious queries from the frontend website are not able to perform any sort of remote code execution (RCE) on the backend database. To ensure this doesn't happen, all inputs from the user need to be sanitized. Testing plans for security testing include performing various types of penetration testing on the frontend, including NOSQL injection and cross-site scripting, by performing queries with malformed input to execute malicious code on the server or in the user's browser.

5.8 Results

The testing on the scraper had two testing requirements associated with it. All tests (100%) had to pass, and the test coverage of scraper Python source code had to be at least 75%. Our unit testing successfully achieved those goals; all unit tests passed, and the coverage is 76%. This was able to show that the scraper is working exactly as intended taking input from the source list and outputting the article data. The following figure shows an output of test coverage in our current implementation of the scraper. Note the total coverage is shown at the top.

Scraper

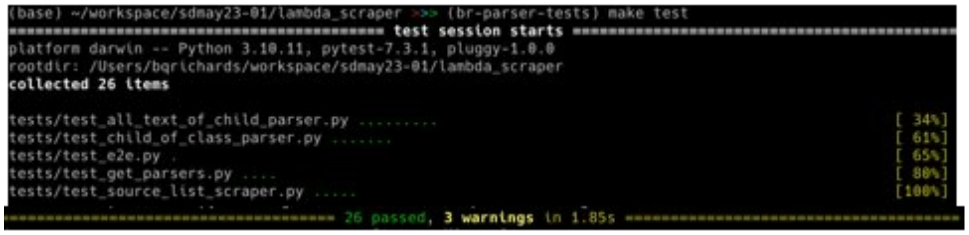


Coverage report: 92%

coverage.py v7.2.3, created at 2023-04-29 11:28 -0500

Module	*statements*	*missing*	*excluded*	*coverage*
AllTextOfChildParser.py	26	0	0	100%
ChildOfClassParser.py	21	0	0	100%
__init__.py	10	0	0	100%
abstract.py	11	0	5	100%
bootstrap.py	25	1	0	96%
dynamo_output.py	21	11	0	48%
source_list_scraper.py	63	11	0	83%
tests/__init__.py	0	0	0	100%
tests/conftest.py	4	0	0	100%
tests/sites_server.py	22	0	0	100%
tests/test_all_text_of_child_parser.py	45	1	0	98%
tests/test_child_of_class_parser.py	42	1	0	98%
tests/test_e2e.py	38	0	0	100%
tests/test_get_parsers.py	17	1	0	94%
tests/test_source_list_scraper.py	39	3	0	92%
Total	**384**	**29**	**5**	**92%**

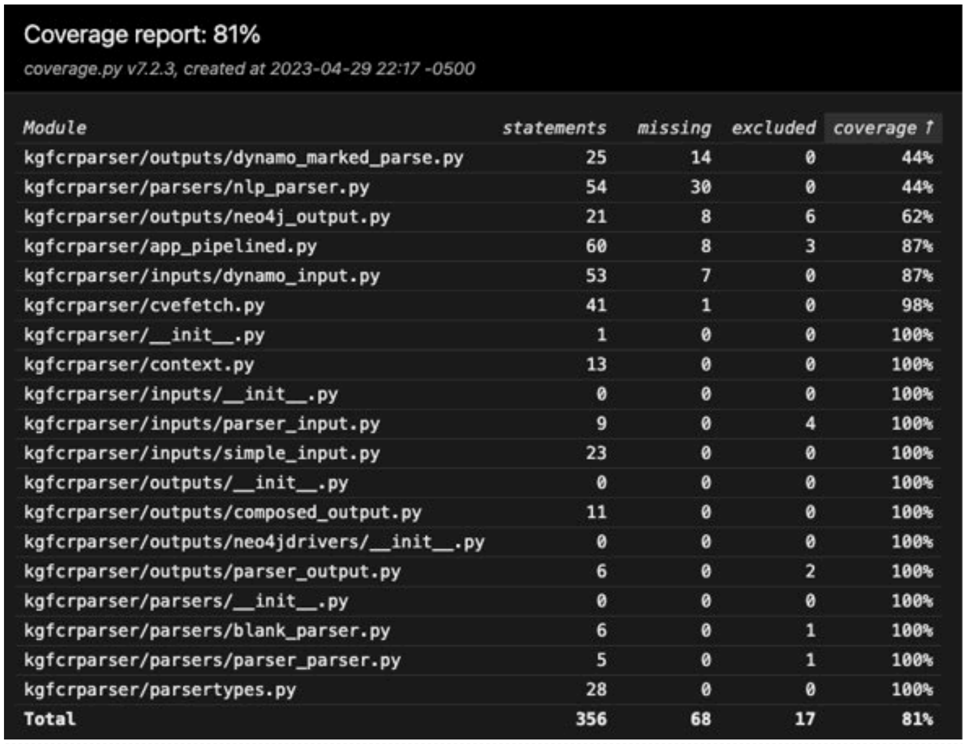
coverage.py v7.2.3, created at 2023-04-29 11:28 -0500



```
(base) ~/workspace/sdmay23-01/lambda_scraper >>> (br-parser-tests) make test
========================================== test session starts ===========================================
platform darwin -- Python 3.10.11, pytest-7.3.1, pluggy-1.0.0
rootdir: /Users/bqrichards/workspace/sdmay23-01/lambda_scraper
collected 26 items

tests/test_all_text_of_child_parser.py .........                                                  [ 34%]
tests/test_child_of_class_parser.py .......                                                       [ 61%]
tests/test_e2e.py .                                                                               [ 65%]
tests/test_get_parsers.py ....                                                                    [ 80%]
tests/test_source_list_scraper.py .....                                                           [100%]
==================================== 26 passed, 3 warnings in 1.85s =====================================
```

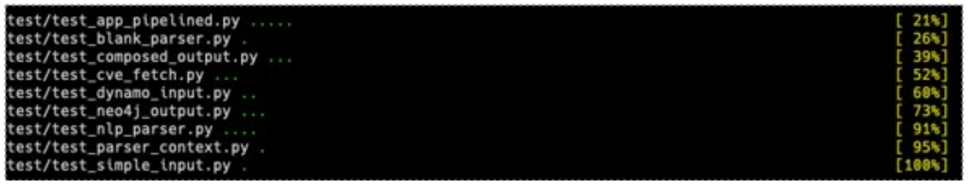
Parser



Coverage report: 81%

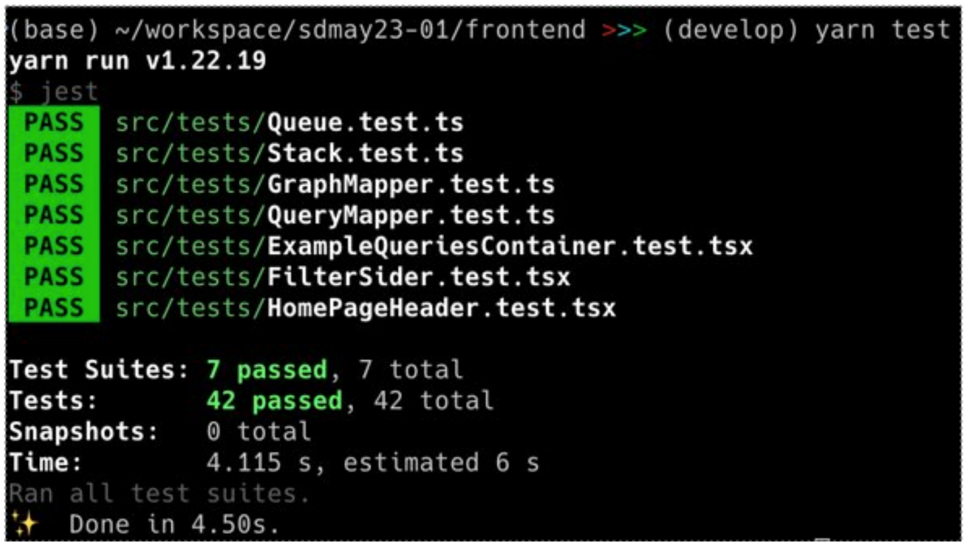
coverage.py v7.2.3, created at 2023-04-29 22:17 -0500

Module	statements	missing	excluded	coverage ↑
kgfcrparser/outputs/dynamo_marked_parse.py	25	14	0	44%
kgfcrparser/parsers/nlp_parser.py	54	30	0	44%
kgfcrparser/outputs/neo4j_output.py	21	8	6	62%
kgfcrparser/app_pipelined.py	60	8	3	87%
kgfcrparser/inputs/dynamo_input.py	53	7	0	87%
kgfcrparser/cvefetch.py	41	1	0	98%
kgfcrparser/__init__.py	1	0	0	100%
kgfcrparser/context.py	13	0	0	100%
kgfcrparser/inputs/__init__.py	0	0	0	100%
kgfcrparser/inputs/parser_input.py	9	0	4	100%
kgfcrparser/inputs/simple_input.py	23	0	0	100%
kgfcrparser/outputs/__init__.py	0	0	0	100%
kgfcrparser/outputs/composed_output.py	11	0	0	100%
kgfcrparser/outputs/neo4jdrivers/__init__.py	0	0	0	100%
kgfcrparser/outputs/parser_output.py	6	0	2	100%
kgfcrparser/parsers/__init__.py	0	0	0	100%
kgfcrparser/parsers/blank_parser.py	6	0	1	100%
kgfcrparser/parsers/parser_parser.py	5	0	1	100%
kgfcrparser/parsertypes.py	28	0	0	100%
Total	**356**	**68**	**17**	**81%**



```
test/test_app_pipelined.py .....                                    [ 21%]
test/test_blank_parser.py .                                         [ 26%]
test/test_composed_output.py ...                                    [ 39%]
test/test_cve_fetch.py ...                                          [ 52%]
test/test_dynamo_input.py ..                                        [ 60%]
test/test_neo4j_output.py ...                                       [ 73%]
test/test_nlp_parser.py ....                                        [ 91%]
test/test_parser_context.py .                                       [ 95%]
test/test_simple_input.py .                                         [100%]
```

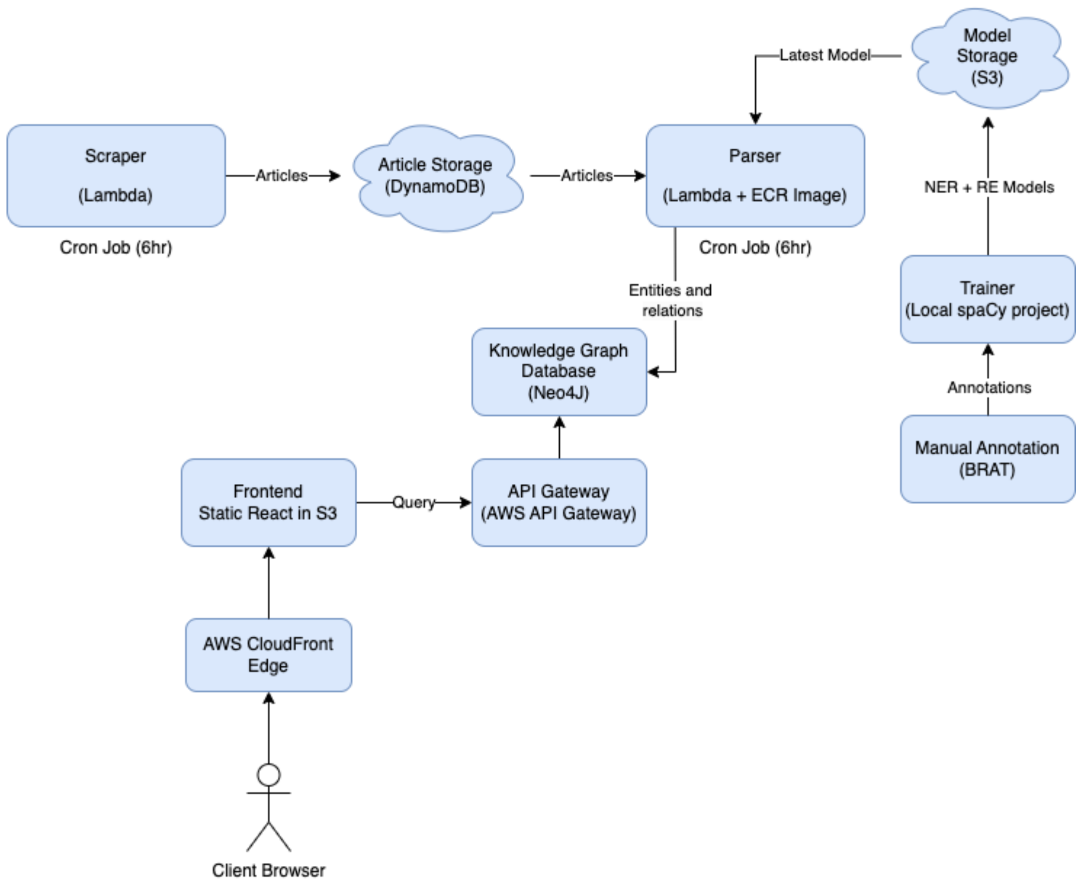
Frontend



```
(base) ~/workspace/sdmay23-01/frontend >>> (develop) yarn test
yarn run v1.22.19
$ jest
 PASS  src/tests/Queue.test.ts
 PASS  src/tests/Stack.test.ts
 PASS  src/tests/GraphMapper.test.ts
 PASS  src/tests/QueryMapper.test.ts
 PASS  src/tests/ExampleQueriesContainer.test.tsx
 PASS  src/tests/FilterSider.test.tsx
 PASS  src/tests/HomePageHeader.test.tsx

Test Suites: 7 passed, 7 total
Tests:       42 passed, 42 total
Snapshots:   0 total
Time:        4.115 s, estimated 6 s
Ran all test suites.
✨  Done in 4.50s.
```

6 Implementation



Infrastructure Architecture

Articles are obtained from our chosen sources and manually annotated for Named Entities and relations between them by members of the team in a software called BRAT. BRAT runs in a Docker container and exposes a web interface to perform the annotations, saving them to the disk. These annotations are then checked into our repo and converted into a .jsonl format which can be used as input to our trainer.

The trainer component is written in Python and is in the form of a spaCy project. The first step performed by the trainer is reading in the .jsonl annotations and converting them to a spaCy-format .spacy binary file. This is done by parsing the input file for the list of annotated entities and relations, adding them to a spaCy Doc object, and adding that object to a spaCy DocBin that can be finally be written as a .spacy binary file. The trainer selects a random 20% of documents to be used as "test" articles. These are not used to train the model, but rather run the model on after training to compare the model's output to our annotator's gold-standard annotations. The trainer first trains the NER model, then the RE model. These are output along with scores of the model, all of which is then uploaded to S3 to be used by the parser.

The scraper is a component written in Python. It makes use of the scrapy library to scrape from a “sourcelist” stored in a JSON format. For each source, scrapy first scrapes the homepage, then each article linked on the home page. Before scraping the article, the scraper queries the article database (DynamoDB) to check if this article has already been scraped. This is done by checking if the URL of the article exists in the database. If it exists, the scraper skips the article. If not, it is scraped and then the record is output into the article database. A record in this database has the following structure:

- url: string
- parsed: boolean
- parsed_version: string
- scrape_time: string
- scrape_version: string
- text: string
- title: string

The parsed field defaults false and parsed_version defaults to an empty string. These fields are set later by the parser. The scraper runs on AWS Lambda and has a schedule to execute every 6 hours.

The parser component is written in Python. It starts by spawning three types of threads: input, parser, and output. The input thread scans the article database in chunks and adds articles that haven’t been parsed to an input queue. There are one or more parser threads monitoring the queue and once they get an article, parsing the text for entities and relations. This is done by using spaCy to load the NER and RE models we train. After parsing the article, the article and tuple of entities and relations are placed into the output queue. There are one or more output threads monitoring this queue and once they obtain a record, they perform an output task with it. We have one output class that writes the entities and relations to our knowledge graph and another class that marks the article as parsed in the article database.

The knowledge graph component is a Neo4J graph database running on an AWS EC2. It exposes an API for both inserting and querying the graph. It also contains the constraints that nodes of a certain type must have unique names. For example, this means there is only one ORGANIZATION node in the graph with the name ‘Microsoft’. There is also an AWS API Gateway that serves as an HTTPS API gateway for the Neo4J EC2. Requests made to this API gateway are forwarded to Neo4J.

The frontend is written in React and hosted as a static site on AWS S3 with an AWS CloudFront edge. CloudFront handles edge computing and providing a domain name, which then serves static content from S3. The frontend queries the knowledge graph using the API gateway, getting back a list of nodes and edges to display on a graph. The query is built using a GUI graph editor built on Uber’s react-digraph library, which is then converted into a Cypher query using a BFS algorithm. The returned nodes and edges are displayed using the vis.js library.

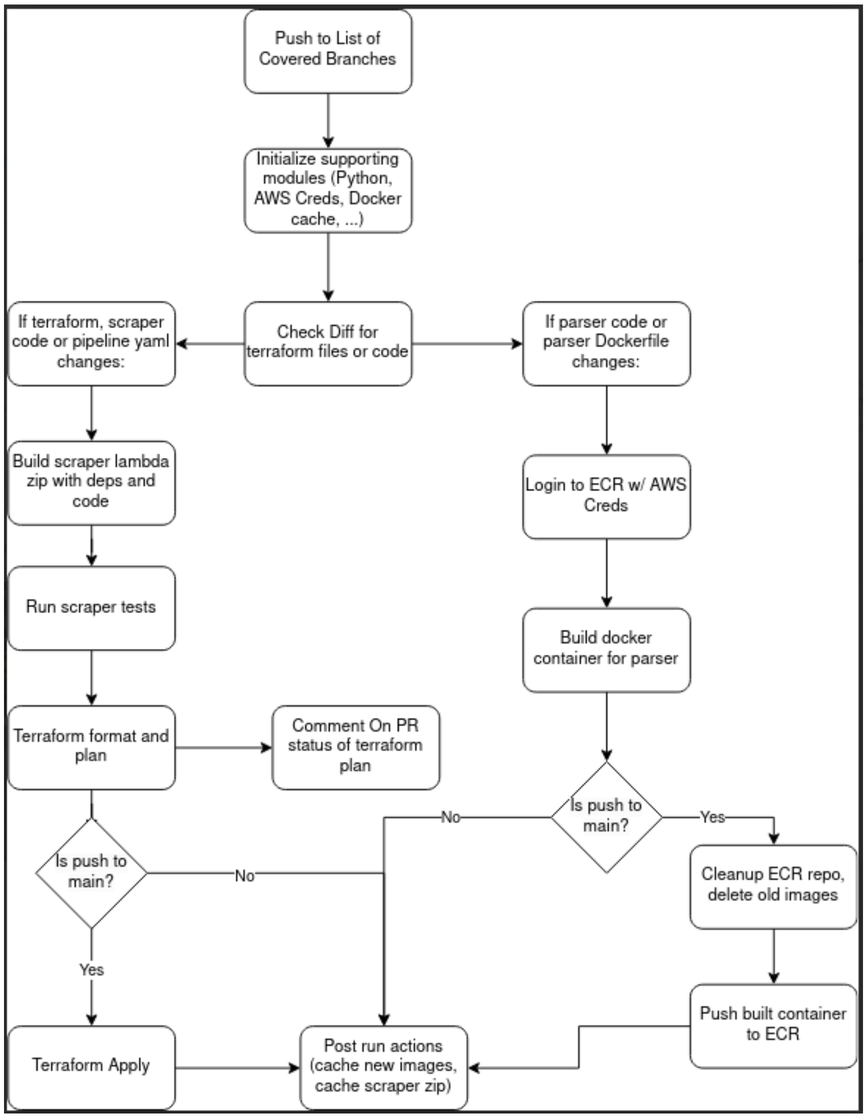
Our code is stored in a GitHub repo with GitHub Actions as our CI/CD pipeline. Every time a pull request or push is made, the pipeline checks to see if a change was made to any of our individual components. If so, it builds, run tests, and then (if pushing to the main branch) deploys the new code. The new code is pushed to AWS using Terraform, which allows us to write Infrastructure-as-Code and associate our built artifacts with certain AWS services.

6.1 EVOLUTION OF DESIGN

This project’s design has evolved quite a lot since 491. In the beginning each module was divided up into docker containers that were ran locally, and the code and docker files were pushed to GitHub so each member of the team could use them on different computers. This design functioned well initially but as the complexity of the modules grew and the design struggled with some requirements the team looked for architectural and deployment changes that could resolve these issues. One major change was the move from running everything locally to running and storing most modules in AWS. The other large change was the implementation of the GitHub Actions pipeline that automatically builds code, deploys infrastructure that is configured through terraform, and comments on Pull Requests to help everyone on the team see the build status and infrastructure plan before merging anything. This added multiple major benefits including:

- Code/Infrastructure Safety: The terraform configuration always requires running a plan before applying anything when running in the pipeline. This ensures that if the plan fails with any issues or the outputted configuration is incorrect the pr author can cancel it and fix the issues.
- Quick And Easy Deployment: Using terraform as IAC allows anyone on the team to quickly configure and add AWS resources just by pushing to GitHub and letting the pipeline deploy everything.
- Automatic Building: The move to cloud for many of the project modules led to the team using AWS lambda as a serverless solution. In terraform all lambda functions must point to a source code zip file, so the pipeline would bundle the code and dependencies together whenever the monitored files change on a push.

The diagram for the logic and architecture of the GitHub Actions pipeline is shown below:


Push to List of Covered Branches
Initialize supporting modules (Python, AWS Creds, Docker cache, ...)
Check Diff for terraform files or code
If terraform, scraper code or pipeline yaml changes:
If parser code or parser Dockerfile changes:
Build scraper lambda zip with deps and code
Login to ECR w/ AWS Creds
Run scraper tests
Build docker container for parser
Terraform format and plan
Comment On PR status of terraform plan
Is push to main?
No
Yes
Is push to main?
No
Yes
Cleanup ECR repo, delete old images
Push built container to ECR
Terraform Apply
Post run actions (cache new images, cache scraper zip)

One important feature of the pipeline came around due to a core problem the team ran into when initially switching to GitHub Actions. With the free version we were using, there was a limit of 2000 build minutes per month. This sounded like plenty at first but when attempting to build the parser the amount of time to download and compress the dependencies was making the pipeline take 8-12 minutes per build which severely limits how many builds we can run especially when team members were having to wait that amount of time for Pull Requests that were not infrastructure related at all. This is where the path action (also referred to as GitHub Diff action) comes in. It allows us to specify variables that are linked to file or directory paths that are true/false based on if the commit the pipeline is running on has changes in those files. In addition, the pipeline can contain conditional statements that recognize whether it is in a Pull Request or Push event and which branch it is running on. This was any PR or push to the “develop” branch for experimenting will not affect the infrastructure in AWS, until that terraform code is Pushed to the “main” branch at which point the Terraform Apply (Production) step is run and/or the build parser container is pushed to ECR.

8 Closing Material

8.1 Conclusion

The project has made significant progress and achieved many of its original objectives. The team successfully generated a Cybersecurity Knowledge Graph, leveraging machine learning techniques to create and train a model off of annotated cybersecurity news articles. The team was able to aggregate information from various sources and meet additional stretch goals that were deemed important for operation. The implementation of querying the Knowledge Graph from the frontend and deploying it to the AWS cloud infrastructure has enhanced accessibility and scalability. Furthermore, by setting up an automated process that runs every 6 hours, the system ensures the continuous update of data. The use of modular interfaces in components has contributed to the system's flexibility and maintainability. Looking ahead, future enhancements could include expanding the article dataset for training to improve the accuracy of Named Entity Recognition (NER) and Relation Extraction (RE), incorporating Natural Language Processing (NLP) for frontend queries, and implementing parser constraints to ensure valid relations. The project could also benefit from an overall improvement of code structure and improvement of the pipeline to help streamline deployment in the cloud infrastructure.

8.2 References

P. Evangelatos, C. Iliou, T. Mavropoulos, K. Apostolou, T. Tsikrika, S. Vrochidis, and I. Kompatsiaris, "Named entity recognition in cyber threat intelligence using transformer-based models," *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021.

P. Ranade, A. Piplai, A. Joshi, and T. Finin, "Cybert: Contextualized embeddings for the cybersecurity domain," *2021 IEEE International Conference on Big Data (Big Data)*, 2021.

N. Rastogi, S. Dutta, A. Gittens, and M. Zaki, "TINKER: A framework for Open source Cyberthreat Intelligence," 21st International Conference on Trust, Security and Privacy in Computing and Communications, Oct. 2022.

8.3 Appendices

8.3.1 Operation Manual

The first step to working with the project is cloning the repo. You'll need to obtain the repo URL and credentials from an authorized user, after which you may clone using the *git clone <URL>* command.

8.3.1.1 Brat

Annotation of articles is done using the tool BRAT. BRAT runs in a Docker container and exposes a web interface. To start the Docker container:

1. Open a terminal
2. Change directory into sdmay23-01/brat

3. Run make build
4. Run make start
5. Open http://localhost:8001 in a web browser
6. Begin annotating

After performing annotations, the *annotations.jsonl* file also needs to be regenerated. Note that this can only be done on a Unix machine, i.e., this is not applicable for Windows.

On a Unix machine, perform the following steps:

1. Open a terminal
2. Change directory into sdmay23-01/trainer
3. Run the command: *./import_brat_to_assets.sh*

The annotations have now been converted to a format spaCy can interpret. Add the *brat/data* directory and the *trainer/assets/annotations.jsonl* to your git stage and follow the steps in the section "How to Contribution to the Repository" to contribute these changes.

8.3.1.2 Trainer

The trainer is located in the *trainer* directory. To perform training, you need to be on a Windows 10 machine and have Anaconda installed. Anaconda is used as a dependency manager for the training project. After this, install the Anaconda environment with the command:

conda env create -f environment.yml

After this, activate this environment with the command:

conda activate sdmay23-01_trainer

This command prompt will now have all the dependencies to train the models. To train the models with a GPU and transformers, run the following command:

make all_gpu

This will take approximately one hour, after which there will be two models in the *training* directory, along with some score .txt files and a hash of the annotations file. Now you have to push these files to AWS S3 so they can be used by the parser when performing information extraction on article text. To upload these newly-trained models, scores, and hash, run the following command:

python scripts\upload_models.py

The latest model has now been pushed to AWS S3.

8.3.1.3 Scraper

The scraper is stored in the *lambda_scraper* folder and the entrypoint to the application is the *src/bootstrap.py* file. After making changes to the scraper, write a test in the *tests* folder to confirm your feature is working and then run the command *make test* to run unit and E2E tests. This will also ensure your changes don't break existing functionality. After your PR is merged to the main branch (please see section: "How to Contribute to the Repository" for more information), you can invoke the scraper Lambda in the AWS Console by the following steps:

1. Log into the senior design AWS account
2. Navigate to the Lambda -> Functions page
3. Click "senior_project_scraper_lambda"
4. View code in the "Code" tab
5. Navigate to the "Test" tab
6. Click the "Test" button at the top right

The scraper will now be invoked and the logs will display after execution is complete.

8.3.1.4 Parser

Before the parser can be run, you will need to obtain access to sensitive credentials. These include:

1. AWS_ACCESS_KEY
2. AWS_SECRET_ACCESS_KEY
3. NEO4J_URL
4. NEO4J_USER
5. NEO4J_PASSWORD

These secrets are needed to interact with the AWS DynamoDB article storage and the Neo4J knowledge graph. After obtaining these values, you will need to create a file in the *parser* directory called *.env*. This file should follow the format:

```
AWS_ACCESS_KEY=<value>

AWS_SECRET_ACCESS_KEY=<value>

NEO4J_URL=<value>

NEO4J_USER=<value>

NEO4J_PASSWORD=<value>
```

After these values are obtained and the environment file is filled, the parser can be run locally by using the *docker-compose.yml* file in the root of the sdmay23-01 directory. Simply navigate to this directory in your terminal and run *docker compose up parser*. This will first build the Docker image for the parser, and then listen and wait for an invocation. This is because it is emulating how it would be waiting for an event in AWS Lambda. To invoke the parser, run the following command:

```
curl -XPOST "http://localhost:9000/2015-03-
31/functions/function/invocations" -d '{}'
```

8.3.1.5 Knowledge Graph

The knowledge graph is hosted on AWS in an EC2 instance. To work directly with the knowledge graph, you will need to log into the senior design AWS account.

1. Log into the senior design AWS account
2. Navigate to the EC2 -> Instances page

From here you can obtain the public domain name of the EC2 and use this to SSH into the machine. You will also need to obtain the confidential *.pem* key to SSH into the machine.

8.3.1.6 Frontend

The frontend is a React app that is built to a static site. To work on the frontend, first change directory into the frontend folder, then install dependencies using the *yarn* command. After installing the dependencies, a development server can be run using *yarn start*. To run tests, use the *yarn test* command. Finally, to build the React app into a static website, run *yarn build*. This generates the site in the *build* directory.

To use the frontend, open a web browser and go to https://d1mthcxyqt7zjw.cloudfront.net/ to use our production frontend, to http://localhost:3000 if you are using the development server, or to the site where you are hosting your own build to use that. You can either manually enter a query in the search bar at the top or use the visual query builder to construct a pattern graph for it to match. To use the visual query builder, click the "Visual Query Builder" button on the left sidebar. You can add nodes to your query by selecting the type from the dropdown menu on the left, optionally adding a name to match (e.g., ORGANIZATION, Microsoft), and clicking the "Add" button. To add a relation between two nodes in your query, hold the shift key and click and drag from one to the other, and select the type of relation if prompted. Once you are done, click OK. Once you have entered your query either manually or using the visual query builder, the resulting graph will be displayed in the main area of the page.

8.3.1.7 Contribution

How to Contribute to the Repository:

When making changes to code or infrastructure as code in the repository it is important to first make a new branch from develop in order to keep the repository organized. Make the changes to main.tf in the root folder for infrastructure or in the appropriate folder for the module you are contributing to. Once the changes are complete, make a Pull Request from your branch back to develop and wait for the pipeline to run. It will output a comment in the PR with the status of the terraform plan and be green if all the pipeline steps are successful and it is ready to be merged. Once the code/infrastructure is in develop and fully tested, a PR can be created from develop to main that contain the changes to be applied to the infrastructure in AWS. The pipeline will then deploy all changed resources and the process is complete.

8.3.2 Alternative Versions of the Design

In the early designs of our project a stretch goal of ours was to implement natural language queries by using natural language processing on the front-end. We ended up not implementing this portion of the project because we did not have the time to work on this feature as we wanted to focus our efforts on the main part of the project, that being to generate accurate knowledge graphs from our articles.

We also had a previous version of how the knowledge graph would look, it even had different nodes and relations and connected in different ways. Upon further digging, we realized that many aspects of the original design were not going to work, as it would eventually structure itself incorrectly, so we had to redesign the graph to be simpler yet more robust. The current version is basically that version, with some additional relations to better help the machine learning model understand how things could be related.

At one time we had also planned to deploy the parser in a lambda, and had spent a large chunk of time preparing to get it to deploy. As we started to finalize things, because of the limited zip upload size of the free version of AWS, we found out that the parser dependencies were much larger than the 250 MB limit and sat at a hefty 4 GB, which was much more than it was willing to handle. We had to scrap the idea and had to make our own custom lambda container to provide the dependencies instead (which had a 10 GB limit), so the project still proceeded but with a minor setback in how it was implemented. The pipeline also had to be modified to build and deploy this container in a reasonable amount of time if the code or dependencies change.

8.3.3 Other Considerations

Because of the nature of training a machine learning model, we encountered a few odd things that happened as the MLM started to parse and fill the Neo4J database with nodes and relations. The team's personal favorite was "oops all versions", where for some reason, the parser decided almost every word it scanned in from an article was a version, and once that information was sent to the graph, it was populated with a few dozen nodes with random words that had been flagged as a version. This was fixed in the next parser update, but it made us wonder where it learned or even what made the parser do that. We also found a whole paragraph from an article that was describing in detail how a distributed denial-of-service (DDoS) attack that was somehow flagged to be vulnerability, even though all of the articles we trained it on didn't have more than three or four words that were an entity. These errors are the reason we mentioned a hefty increase of the items the model can be trained on, as seeing a few hundred more examples would allow it to learn better and then perform its own NER and RE more accurately.