I have taken the rules for all the files:

*Processing file: rules/emerging-netbios.rules*

***392 new rules appended to extracted\_rules.json. Total rules: 40576***

*Last extracted SID: 2103157*

Now I have skipped the # alerts here as they are commented out from the .rules and apart from that all the /rules files having the .rule have been extracted by me.

I created a script that extracts this here’s its documentation:

**Documentation: Suricata Rule Extractor Script**

**Overview**

The **Suricata Rule Extractor Script** is a Python program designed to parse Suricata .rules files, extract specific fields (suri\_rule\_id, suri\_rule\_classtype, and suri\_rule\_msg), and save the results to a consolidated JSON file. This script supports both individual .rules files and bulk processing of all .rules files within a specified directory.

**Features**

1. **Field Extraction**:
   * Extracts key fields from Suricata rules:
     + **suri\_rule\_id**: Unique ID of the rule (sid field).
     + **suri\_rule\_classtype**: Classification type of the rule (classtype field).
     + **suri\_rule\_msg**: Human-readable message describing the rule (msg field).
2. **Deduplication**:
   * Ensures no duplicate suri\_rule\_id entries are added to the output JSON file.
3. **Single File and Bulk Processing**:
   * Process a single .rules file by providing its path.
   * Process all .rules files within a folder (rules/) when bulk processing is enabled.
4. **Configurable Activation/Deactivation**:
   * A flag (PROCESS\_ALL\_FILES) allows you to toggle between single file processing and bulk directory processing.
5. **Error Handling**:
   * Skips and logs lines that do not conform to the expected format.
   * Safely handles partially corrupted or empty files.
6. **Logs and Summaries**:
   * Provides informative logs about:
     + The number of new rules appended.
     + Duplicate rules that were skipped.
     + The last extracted suri\_rule\_id.

**Script Components**

1. **extract\_fields\_from\_rules()**:
   * Parses a .rules file line by line.
   * Uses regular expressions to extract msg, classtype, and sid.
   * Skips empty lines and commented-out rules (#alert).
2. **append\_to\_json\_file()**:
   * Loads an existing JSON file (if it exists).
   * Appends new rules while checking for duplicates based on suri\_rule\_id.
   * Writes the updated data back to the JSON file.
3. **process\_all\_rules\_in\_folder()**:
   * Loops through all .rules files in a specified folder.
   * Calls extract\_fields\_from\_rules() and append\_to\_json\_file() for each file.
4. **Main Execution Block**:
   * Uses the PROCESS\_ALL\_FILES flag:
     + True: Processes all .rules files in the rules/ folder.
     + False: Processes a single .rules file provided as a command-line argument.

**How the Script Works**

1. **Setup**:
   * Place all .rules files in a folder named rules/.
   * Ensure the script (suricata\_rule\_extractor.py) is in the same directory.
2. **Configuration**:
   * Open the script and set the PROCESS\_ALL\_FILES flag:
     + True to process all .rules files in the rules/ folder.
     + False to process a single file.
3. **Running the Script**:
   * **To Process All Files**: Run the script without any arguments:

python suricata\_rule\_extractor.py

* + **To Process a Single File**: Run the script with the file path as an argument:

python suricata\_rule\_extractor.py rules/3coresec.rules

1. **Output**:
   * The script creates or updates a file named **extracted\_rules.json**.
   * Each rule is saved in the following JSON format:

[

{

"suri\_rule\_id": "2525000",

"suri\_rule\_classtype": "misc-attack",

"suri\_rule\_msg": "ET 3CORESec Poor Reputation IP group 1"

},

{

"suri\_rule\_id": "2525001",

"suri\_rule\_classtype": "misc-attack",

"suri\_rule\_msg": "ET 3CORESec Poor Reputation IP group 2"

}

]

1. **Logging**:
   * The script outputs logs such as:
     + Processing file names.
     + Skipped duplicates.
     + Total new rules are appended.
     + The last suri\_rule\_id processed.

**Development Steps**

1. **Initial Script**:
   * Created a script to process a single .rules file.
   * Extracted msg, classtype, and sid using a regular expression.
2. **Handling Commented-Out Rules**:
   * Ensured that rules starting with # (comments) were ignored.
3. **Deduplication**:
   * Added a check to prevent duplicate suri\_rule\_id entries in the JSON file.
4. **Appending to JSON**:
   * Modified the script to append data to an existing JSON file without overwriting it.
5. **Bulk Processing**:
   * Developed a function process\_all\_rules\_in\_folder() to loop through all .rules files in the rules/ directory.
6. **Activation/Deactivation Flag**:
   * Introduced the PROCESS\_ALL\_FILES flag to allow toggling between single-file and multi-file processing.
7. **Logging and Error Handling**:
   * Added informative logs for skipped duplicates, malformed lines, and processing summaries.

***50 rulefiless in total***

***And 50 MITRE ATT&CK TECHNIQUES***

Suriclasstype meaning and deinitions:

# config classification:shortname,short description,priority

config classification: not-suspicious,Not Suspicious Traffic,3

config classification: unknown,Unknown Traffic,3

config classification: bad-unknown,Potentially Bad Traffic, 2

config classification: attempted-recon,Attempted Information Leak,2

config classification: successful-recon-limited,Information Leak,2

config classification: successful-recon-largescale,Large Scale Information Leak,2

config classification: attempted-dos,Attempted Denial of Service,2

config classification: successful-dos,Denial of Service,2

config classification: attempted-user,Attempted User Privilege Gain,1

config classification: unsuccessful-user,Unsuccessful User Privilege Gain,1

config classification: successful-user,Successful User Privilege Gain,1

config classification: attempted-admin,Attempted Administrator Privilege Gain,1

config classification: successful-admin,Successful Administrator Privilege Gain,1

# NEW CLASSIFICATIONS

config classification: rpc-portmap-decode,Decode of an RPC Query,2

config classification: shellcode-detect,Executable code was detected,1

config classification: string-detect,A suspicious string was detected,3

config classification: suspicious-filename-detect,A suspicious filename was detected,2

config classification: suspicious-login,An attempted login using a suspicious username was detected,2

config classification: system-call-detect,A system call was detected,2

config classification: tcp-connection,A TCP connection was detected,4

config classification: trojan-activity,A Network Trojan was detected, 1

config classification: unusual-client-port-connection,A client was using an unusual port,2

config classification: network-scan,Detection of a Network Scan,3

config classification: denial-of-service,Detection of a Denial of Service Attack,2

config classification: non-standard-protocol,Detection of a non-standard protocol or event,2

config classification: protocol-command-decode,Generic Protocol Command Decode,3

config classification: web-application-activity,access to a potentially vulnerable web application,2

config classification: web-application-attack,Web Application Attack,1

config classification: misc-activity,Misc activity,3

config classification: misc-attack,Misc Attack,2

config classification: icmp-event,Generic ICMP event,3

config classification: policy-violation,Potential Corporate Privacy Violation,1

config classification: default-login-attempt,Attempt to login by a default username and password,2

# Update

config classification: targeted-activity,Targeted Malicious Activity was Detected,1

config classification: exploit-kit,Exploit Kit Activity Detected,1

config classification: external-ip-check,Device Retrieving External IP Address Detected,2

config classification: domain-c2,Domain Observed Used for C2 Detected,1

config classification: pup-activity,Possibly Unwanted Program Detected,2

config classification: credential-theft,Successful Credential Theft Detected,1

config classification: social-engineering,Possible Social Engineering Attempted,2

config classification: coin-mining,Crypto Currency Mining Activity Detected,2

config classification: command-and-control,Malware Command and Control Activity Detected,1

**LLM Experimentation Design:**

**Novel Contribution: Enhancing Suricata Rule-to-MITRE Mapping Using LLMs**

To make a novel contribution using LLMs, we can go beyond simple text matching and focus on advanced techniques that enhance the accuracy, explainability, and scalability of Suricata rule mapping to the MITRE ATT&CK framework. Below are some innovative ideas and corresponding design suggestions.

**1. LLM-Assisted Explainable Mapping**

**Contribution:**  
Instead of just mapping suri\_rule\_msg to a MITRE technique, the LLM can generate explanations for why a specific technique was selected.

**Design:**

* **Prompt Engineering:** Ask the LLM to not only select the MITRE technique but also provide reasoning based on the keywords or patterns detected in the suri\_rule\_msg.

**Example Prompt:**

Suricata Rule Message: "ET 3CORESec Poor Reputation IP group 1"

MITRE ATT&CK Techniques:

- T1071 - Command and Control: Use of external servers for communication

- T1065 - Exploit Public-Facing Application

Task:

1. Map the Suricata rule message to the most relevant MITRE ATT&CK technique.

2. Provide an explanation for your selection based on the keywords or context.

Output Format:

- Technique ID: <ID>

- Technique Name: <Name>

- Explanation: <Reasoning>

**Example Output:**

{

"Technique ID": "T1065",

"Technique Name": "Exploit Public-Facing Application",

"Explanation": "The rule mentions 'Poor Reputation IP', which suggests malicious use of public IPs. This aligns with external applications being exploited for attacks."

}

**2. Hybrid Approach: Rule Contextualization**

**Contribution:**  
Integrate contextual metadata (e.g., classtype, IP addresses, or other metadata) along with suri\_rule\_msg to provide richer inputs for the LLM. This can improve the precision of mapping. **Maybe the entire Suricata rule provision can be helpful?**

**How to Design:**  
Combine multiple Suricata fields (suri\_rule\_msg, suri\_rule\_classtype, and IP/port details) into a single structured input for the LLM.

**Example Input:**

Suricata Rule:

- Message: "ET 3CORESec Poor Reputation IP group 1"

- Classification: "misc-attack"

- Context: External IP address targeting HOME\_NET.

MITRE ATT&CK Techniques:

<List of techniques and descriptions>

Task: Map the rule message and its classification to the most appropriate MITRE ATT&CK technique.

This approach uses the classtype and other metadata to provide additional context, helping the LLM make better decisions.

**3. LLM Fine-Tuning for Cybersecurity Context**

**Contribution:**  
Train a custom fine-tuned LLM on Suricata rules and MITRE ATT&CK mappings. Fine-tuning ensures that the model understands cybersecurity-specific vocabulary and mappings more effectively than a generic LLM. But then having mappings that are consistent that dataset is not really available to us now

**How to Design:**

1. **Dataset Creation:**
   * Gather a large dataset of Suricata rules mapped to MITRE techniques (manually or semi-automatically curated).
   * Format the data as input-output pairs:
     + **Input:** Suricata rule fields (msg, classtype, etc.).
     + **Output:** MITRE technique (technique\_id, technique\_name, explanation).
2. **Fine-Tuning:**
   * Use OpenAI's fine-tuning API (or Hugging Face for open models) to train the model on this dataset.
   * Evaluate the fine-tuned model against unseen Suricata rules to measure accuracy.

**Example Input-Output Pair for Fine-Tuning:**

**Input:**

{

"suri\_rule\_msg": "ET POLICY SSH Brute-Force Login Attempt",

"suri\_rule\_classtype": "attempted-user"

}

**Output:**

{

"technique\_id": "T1110",

"technique\_name": "Brute Force",

"explanation": "The rule message describes an SSH brute-force attempt, which matches the MITRE ATT&CK brute-force technique."

}

**4. Confidence Scoring for Mapping(List of relevant mappings provide at the each prediction)**

**Contribution:**  
Generate a confidence score for each mapping to indicate how certain the LLM is about its decision. This helps analysts prioritize rules that need manual verification.

**How to Design:**

* Use the LLM to generate additional output: a confidence score based on the strength of keyword matches or reasoning.
* Scale the score between 0 and 1 (or as a percentage).

**Prompt Example:**

Suricata Rule Message: "ET POLICY SSH Brute-Force Login Attempt"

MITRE ATT&CK Techniques:

1. T1110 - Brute Force

2. T1071 - Command and Control

Task:

1. Select the most relevant MITRE ATT&CK technique.

2. Provide a confidence score (0-1) based on the strength of match.

Output Format:

- Technique ID: <ID>

- Technique Name: <Name>

- Confidence Score: <0-1>

- Explanation: <Reasoning>

**Example Output:**

{

"Technique ID": "T1110",

"Technique Name": "Brute Force",

"Confidence Score": 0.95,

"Explanation": "The rule mentions 'Brute-Force Login Attempt', which strongly matches the MITRE technique for brute force."

}

**5. Multi-LLM Ensemble for Validation**

**Contribution:**  
Use multiple LLMs (e.g., GPT-4, open-source models like Llama2, or fine-tuned models) to independently map rules to MITRE ATT&CK techniques. Aggregate their responses for a more reliable mapping. Like a set of experimentations

**How to Design:**

* Use an ensemble approach where:
  + Multiple LLMs generate a mapping.
  + Compare outputs and pick the most frequently selected technique.
  + If discrepancies arise, flag them for manual review.

**Output Example:**

{

"suri\_rule\_id": "2525000",

"suri\_rule\_msg": "ET 3CORESec Poor Reputation IP group 1",

"mapped\_techniques": [

{"LLM": "GPT-4", "Technique ID": "T1065", "Confidence": 0.90},

{"LLM": "Llama2", "Technique ID": "T1071", "Confidence": 0.85}

],

"final\_technique": "T1065"

}

Now the set of experiments we will perform on the first 100 suricata rulesets to selct which one is the best then how do we test the LLM outputs? How do INkow they perform well base don the outputs do we need a human INTERVENTION FOR TESTING OR IS THERE SOME OTHER METRIC THAT WE CAN USE AS A RECONICLATION STEP?

What I have found out is that the LLM are able to predict well.

The questions are where do we test it, how do we test it, running them in a sample size should be the best at each llm session giving a Bigger context size with a MITRE Technique ma[[ing should again be the best approach.

Should we store the MITRE Techniques as embedding and make a retrieval based on the similarity score? I guess no that wouldn’t be a good approach. But can we try embedding retrieval for the top 5 and then put it in LLM to suggest the best way to save us some space and context size?

Then test it multiple times to ensure outputs are more or less similar.

Do a set of experiments to find the optimal batch size for the outputs that are repeatedly the same.

In the set of 10 experiments that I did the outputs were similar.

Cost matrix, Cron job- Assume it is running just once as of now.

Make a documentation, slides,

Conduct experiments with 10 epochs and 100 epochs record the cost vs API usage,

Breaking points

**1. Experiment Setup**

**1.1 Initial Dataset Selection**

* Select the first **10** or **100 Suricata rules** as the sample set.
* Ensure each rule contains the suri\_rule\_msg and corresponding MITRE mapping (ground truth).

**1.2 Experimental Goals**

* Evaluate the **accuracy** and **consistency** of LLM outputs.
* Measure **cost-efficiency** based on API usage.
* Identify the **optimal batch size** for processing rules.
* Identify the **optimal prompts** for the optimisation in terms of context size-(,)
* Ensure outputs are **stable and repeatable** across multiple runs.
* Identify the **different models, their context window sizes** and the trade-offs between them in terms of **Cost vs API** calls, accuracy etc.
* Make a cost estimation table for the job that runs once in terms of 40,000 rules processed in chunks and batches.
* Cost vs API for passing the MITRE Technique in every prompt due to the assumption problem the model is going through at present. This would also increase the context size required.
* Script for model output vs the model accuracy basically uses Jonathan’s dataset as the test set.

First do the set of experiments on **10 rules** vs (**Prompts, Batch size, Context Window Size, Cost vs API**).

**2. LLM Testing Framework**

**2.1 Experiment Phases**

* **Phase 1: Baseline Testing**
  + Run the LLM on the first 10 rules individually.
  + Record outputs for each rule.
  + Compare predictions against the **ground truth** MITRE mappings.
  + Metrics to record:
    - Prediction Accuracy: % of correct MITRE techniques.
    - Consistency: Similar outputs across multiple runs.
    - Context Size Used: Evaluate token consumption.
* **Phase 2: Batch Size Optimization**
  + Process rules are in batches of **10 and 20 (50 and 100 are currently unavailable)**.
  + Record results:
    - Accuracy of predictions for each batch size.
    - API cost for each batch size.
    - Identify the optimal batch size with minimal token usage and consistent accuracy.
* **Phase 3: Similarity Testing with Embeddings**
  + Store MITRE techniques as **embeddings** (vector format).
  + Retrieve the **top 5 similar techniques** for each Suricata rule using a similarity score.
  + Pass these techniques to the LLM and ask it to select the most relevant one.
  + Compare the accuracy and cost:
    - Is this method reducing **context size** and **cost**?
    - Are results comparable to the baseline?

**3. Testing Metrics**

**3.1 Key Metrics**

* **Prediction Accuracy:** % of correct predictions vs ground truth.
* **Consistency:** Repeated experiments with the same inputs yielding the same output.
* **Cost:** API usage cost based on batch size and tokens consumed.
* ***Execution Time***(Maybe not important) Time taken for batch vs individual processing, to times taken by different models at different context sizes.

**3.2 Reconciliation**

* Determine if **human intervention** is required:
  + Review a subset of predictions manually (e.g., 10%).
  + Use accuracy as a reconciliation step.
* Eval-Metrics:
  + **Similarity Score**: Compare LLM output with ground truth using cosine similarity or exact matches.

**4. Experiment Repetition**

**4.1 Stability Check**

* Run the LLM experiments **10 times** to ensure output stability.
* Compare results:
  + % of similar outputs over repeated runs.
  + Identify any variance.

**4.2 Rule-Based Testing**

* Conduct experiments for:
  + **10 rules**: Measure cost, accuracy, and time.
  + **100 rules**: Measure improvements (or diminishing returns).
* Compare:
  + Cost vs API usage.
  + Breaking points (if any).

**5. Documentation and Presentation**

**5.1 Experiment Records**

* Maintain logs for:
  + Inputs (Suricata rules).
  + Outputs (MITRE mappings suggested by the LLM).
  + Accuracy scores, costs, and batch sizes.

**5.2 Slides and Summary**

* Summarize findings in a presentation:
  + Accuracy comparisons.
  + Cost optimization results.
  + Recommendations for best methods (batch size, embeddings, reconciliation metrics).

**6. Automation and Future Considerations**

* Schedule the final pipeline as a **Cron job** (assume it runs once for now).
* Automate the logging and recording of results for scalability.
* Explore further optimizations for larger datasets(40,000 rules in total).

1. **Documentation**:
   * Experiment logs, results, and reconciliation analysis.
2. **Slides**:
   * Summary of findings and recommendations.
3. **Cost Analysis**:
   * Cost matrix for batch sizes, API usage, and breaking points.
4. **Code Scripts**:
   * Scripts for batch processing, embedding retrieval, and evaluation.

**Problems:**

1. I have a small test set for testing. Need a varied test set of around 100-500 different alerts for creating a good test set.
2. The OPENAI API key and account
3. Using the OPEN dataset which has a lesser set of rules so would make sense to decide if we want to go with a Suricata PRO or OPEN set before proceeding with the set of experiments to ensure consistency.
4. Would be better to run the script in batches to ensure a human can review the data in batches of 500-1000 to ensure there’s no loss in terms of the API calls monetarily.

**Experiment Table Format:**

Tables1: ChatGPT Models || Prediction accuracy(Ground-Truth vs LLM-Output)

Tables2: Batch Size Optimisation Table

Tables3: Batch-Size for Rules || Accuracy || Token Size || API Cost || Execution Time

Tables4: Embedding Retrieval + LLM Output

Accuracy vs different GPT models with embedding and without embedding

Tables5: Cost vs Accuracy

For testing with MITRE Technique, without mitre techniques

**Graphs visualization:**

 Accuracy vs. Cost Graph

 Batch Size vs. Token Consumption Chart

Which experiments to prioritize and their prioritizing order:

**Suggested Experimentation Order:**

1. Developing an accuracy measurement script
2. Baseline Testing (ChatGPT Models)
3. Batch Size Optimization of the Number of Suricata Rules
4. Token Size to take
5. Cost vs Accuracy (With and Without MITRE Techniques)
6. Accuracy vs Embedding Retrieval
7. Repeated Runs for Consistency Testing

Graphs for the Batch Size, Token Size and (Cost vs Accuracy)

**Different Types of Rule Files**

There are **55 rule files**, with an expected total of **515 rules**. Below is a breakdown of the individual rule files and the number of rules in each:

1. **Coresec Rules** - 10 rules
2. **Botcc.portgrouped.rules** - 0 rules
3. **Botcc.rules** - 0 rules
4. **Compromised.rules** - 10 rules
5. **Drop.rules** - 10 rules
6. **Dshield.rules** - 1 rule
7. **Emerging-activex.rules** - 19 rules
8. **Emerging-adware\_pup.rules** - 10 rules
9. **Emerging-attack\_responses.rules** - 10 rules
10. **Emerging-chat.rules** - 10 rules
11. **Emerging-coinminer.rules** - 10 rules
12. **Emerging-current\_event.rules** - 10 rules
13. **Emerging-deleted.rules** - 0 rules
14. **Emerging-dns.rules** - 10 rules
15. **Emerging-dos.rules** - 10 rules
16. **Emerging-dyn-dns.rules** - 11 rules
17. **Emerging-exploit\_kit.rules** - 10 rules
18. **Emerging-exploit.rules** - 10 rules
19. **Emerging-file\_sharing.rules** - 10 rules
20. **Emerging-ftp.rules** - 10 rules
21. **Emerging-game.rules** - 11 rules
22. **Emerging-hunting.rules** - 10 rules
23. **Emerging-icmp.rules** - 10 rules
24. **Emerging-imap.rules** - 10 rules
25. **Emerging-inappropriate.rules** - 0 rules
26. **Emerging-info.rules** - 9 rules
27. **Emerging-ja3.rules** - 11 rules
28. **Emerging-malware.rules** - 10 rules
29. **Emerging-misc.rules** - 10 rules
30. **Emerging-mobile\_malware.rules** - 10 rules
31. **Emerging-netbios.rules** - 11 rules
32. **Emerging-p2p.rules** - 10 rules
33. **Emerging-phishing.rules** - 10 rules
34. **Emerging-pop3.rules** - 9 rules
35. **Emerging-remote\_access.rules** - 11 rules
36. **Emerging-retired.rules** - 10 rules
37. **Emerging-rpc.rules** - 10 rules
38. **Emerging-scada.rules** - 10 rules
39. **Emerging-scan.rules** - 10 rules
40. **Emerging-shellcode.rules** - 11 rules
41. **Emerging-smtp.rules** - 10 rules
42. **Emerging-snmp.rules** - 10 rules
43. **Emerging-sql.rules** - 10 rules
44. **Emerging-ta\_abused\_services.rules** - 10 rules
45. **Emerging-telnet.rules** - 8 rules
46. **Emerging-tftp.rules** - 12 rules
47. **Emerging-user\_agents.rules** - 10 rules
48. **Emerging-voip.rules** - 10 rules
49. **Emerging-web\_client.rules** - 11 rules
50. **Emerging-web\_server.rules** - 10 rules
51. **Emerging-web\_specific\_apps.rules** - 10 rules
52. **Emerging-worm.rules** - 9 rules
53. **Threatview\_CS\_c2.rules** - 10 rules
54. **Tor.rules** - 10 rules

**Total Rules**

The total number of rules across all files is **515**.

I mapped all the rules to the LLM Outputs and got a CSV file.

**Meeting Notes: Andrew (MOM)**

**Key Discussion Points**

1. **Performance Evaluation: GPT-3.5 Turbo**
   * Begin by assessing how GPT-3.5 Turbo performs on the current mapping tasks.
   * Compare the performance of GPT-3.5 Turbo with different models to develop the test set:
     + **Accuracy**: Measure accuracy in mapping Suricata rules to MITRE ATT&CK techniques based on the Cyber-SME’s feedback.
     + **Cost-Effectiveness**: Analyse the cost per token for both models to optimize budget usage.
   * Consider cost vs. optimization as a core metric in this evaluation.
2. **Vector Database Integration**
   * Implement a vector database to store the 300+ MITRE ATT&CK techniques and their attributes.
   * Perform look-ups in the vector database for quicker and more precise similarity matching.
   * Experiment with **contextual embeddings** (e.g., via OpenAI or similar) to improve retrieval results.
3. **Chain-of-Thought (CoT) Models**
   * Explore models like **AutoGPT** and **ThinkGPT** for tasks requiring reasoning pipelines.
   * Incorporate chain-of-thought reasoning to decompose complex mapping tasks into smaller steps.
4. **Pipeline with Multiple Agents**
   * Develop a hierarchical model pipeline with three specialized agents:
     + **Testing Model (Ranker)**:
       - Ranks mapping results based on the confidence of relevance to MITRE ATT&CK techniques.
       - Determines most likely and least likely matches.
     + **Evaluator Agent**:
       - Validates the outputs from the testing model.
       - Checks the rankings and mappings against the MITRE ATT&CK database for accuracy.
     + **Correction Agent**:
       - Incorporates feedback from the evaluator agent.
       - Updates and writes the final mappings into JSON format.
   * This pipeline reduces the need for manual intervention by cybersecurity SMEs to some extent.
5. **Experimental Design for Model Pipelines**
   * Define experiments to optimize agent tasks:
     + Assign and measure specific roles for each agent in the pipeline.
     + Test scenarios with varying **hierarchical model designs**.
   * Evaluate whether the pipeline design leads to:
     + **Improved accuracy** of mapping.
     + Reduction in the need for manual corrections.

Ensemble, Learn comine lm outputs to determine threshold

Boosting.

1. **Automated Evaluation Metrics**
   * Implement heatmaps for **model evaluation**:
     + Visualize discrepancies between actual vs. predicted mappings. Us performance metric like precision recall for tracking the performance
2. **Cost Analysis: Token and Parameter Trade-offs**
   * Evaluate cost-effectiveness metrics such as:
     + **$/million input tokens**.
     + **$/million output tokens**.
   * Compare the parameter sizes of models (e.g., GPT-3.5, GPT-4) to understand trade-offs:
     + Smaller models (cheaper) vs. larger models (potentially more accurate).
3. **Hyperparameter Tuning**
   * Ensure fine-tuning of hyperparameters (e.g., temperature, max tokens) for specific tasks.
   * Post-tuning, investigate:
     + The impact of different model types.
     + The role of embeddings in improving performance.

**Action Plan**

1. **Short-Term Goals:**
   * Test the GPT-3.5 Turbo model on a subset of data for performance benchmarking.
   * Begin vector database integration for MITRE techniques.
   * Develop a basic version of the 3-agent pipeline for testing.
2. **Long-Term Goals:**
   * Scale the 3-agent pipeline for full dataset processing.
   * Perform comprehensive cost-vs-performance evaluations across multiple model types (GPT-3.5, GPT-4).
   * Expand the pipeline design to incorporate additional agents if necessary.
3. **Future Experiments:**
   * Investigate **alternative models** for each task in the pipeline based on input complexity and token limits.
   * Refine the vector database and embedding generation for faster look-ups.

Implement the mitre 300 mapping evaluator feedback as well, Copy the 300 MITRE Techniques into a JSON file, then implement the vectorDB. Maybe get the three list of outputs or 4 outputs with their likelihood or probabilities and that can be then posted on the app for the Cyber SME’s.