# Project Proposal

The objective of this project is to deliver a fully functional, end-to-end Python script that can read generated code files, establish classification patterns, and detect anomalies. We will prioritize a runnable MVP (Minimum Viable Product) first, which means that the initial focus should be proving the core concept works

# Problem Description

## Background

We have a link on a web page. There might be another link after you click that link. Then you might have a button to download a file. We assume that we might have downloaded 100 files. Let's call them file1 to file100
We have a program, A.exe. It is taking the 100 files as an input. Then generate 100 corresponding files, called 1.txt, 2.txt, 3.txt, ... 100.txt.

## Problem

We would like to group the output 100 files. By putting them into groups, we can better analyze them. We might be able to access the grouping rules and a sample dataset to group them based on the sample group rules. And as the input file increases, the current grouping method may not be enough. There will be outliers and we need to add new groups to the current groups.
We will use AI models and Python scripts to solve the problem

# Solution

## Assumption

We assume the file download and excution at a.exe is already done/considered to be out of scope

## Process

1. **Read the .txt file**
2. **Extract dataset**
    - Listed some possible solutions of model and made comparation
    - I selected the `Word2Vec/FastText` approach, followed by `Averaging Word Vectors` to create the final document embedding
3. **Model Training & Initial Clustering**
    - DBSCAN Application
    - Cluster Center Calculation
4. **New File Classification & Incremental Model Update**
    - New File Encoding

    - Distance and Threshold Check
    - Decision
5. **Reclustering**
    - Data Preparation and Full Retraining

I have a detailed solution attached at the end of the proposal.

# Executive Summary and Project Goal

The primary goal is to validate the core machine learning concept—combining `Word2Vec` embeddings with density-based clustering—before scaling.

| Component | Goal | Metric |
|---|---|---|
| **Feature Extraction** | Generate robust, fixed-length feature vectors. | $10,000 \times 100$ Feature Matrix ($\mathbf{X}$) generated. |
| **Clustering** | Define stable, initial file types (Groups $\mathbf{C_k}$ ). | $\mathbf{DBSCAN}$ executed; $\mathbf{C_k}$ calculated. |
| **Anomaly Detection** | Establish a reliable threshold for outlier detection. | Validated Anomaly Threshold ($\boldsymbol{\Theta}$) calculated. |

# Execution Plan (MVP Focus)

The focus is on delivering a runnable, end-to-end demonstration. Complex production engineering (e.g., database integration, Airflow) is deferred. The total estimated time is about 2.5 weeks

## Phase 1: Feature Foundation (Est. 1 Week)

- **Action:** Finalize custom tokenizer, handle 10k file reading, and train the full Word2Vec model.
- **Output:** Saved `Word2Vec.model` and the full $\mathbf{10,000 \times 100}$ feature matrix $\mathbf{X}$.

## Phase 2: Core Logic Implementation (Est. 1 Week)

- **Action:** Run DBSCAN, tune $\epsilon/\mathbf{min\_samples}$, calculate all $\mathbf{C_k}$ **centroids**, and determine the final $\boldsymbol{\Theta}$ **threshold**.
- **Output:** Defined cluster metrics and the fully calibrated set of parameters ready for classification.

## Phase 3: Update Cycle and Delivery (Est. 3-5 Days)

- **Action:** Develop the `process_new_file()` classification function (Steps 1-3) and the `perform_full_retraining()` simulation function (Step 5).
- **Output:** The final, runnable `clustering_solution.py` script demonstrating the entire end-to-end

process, including the ability to adapt to new outlier data.

# Scope and Deferred Work

To meet the timeline, the following items are explicitly **excluded** from this MVP and are reserved for a subsequent Production Phase:

- **Performance Optimization:** Advanced parallelization (e.g., GPU/Dask) for matrix generation and retraining.
- **Infrastructure:** Persistent storage (e.g., PostgreSQL or S3) for models and vectors.
- **Deployment:** CI/CD pipelines, containerization (Docker), or dedicated scheduling (Airflow).
- **Extensive Hyperparameter Search:** Tuning is limited to a small, effective range for DBSCAN parameters.

# Possible Timeline for Future Work

When MVP is complete, we will move to our deferred work. We will focus on hardening, scaling, and deploying the solution to handle the large amount of real-time data reliably and continuously.

| Phase | Focus Area | Estimated Duration | Key Deliverables |
|---|---|---|---|
| **Phase 4: Infrastructure & Scalability** | **Data Pipeline and Storage** | 2 - 3 Weeks | **Persistent Storage:** Migrate vectors ($\mathbf{X}$), model files, and centroids ( $\mathbf{C_k}$) from local files to a robust data store (e.g., PostgreSQL/S3).<br>**Optimized I/O:** Implement efficient data loading/saving for the 10k file corpus during retraining.<br>**Model Optimization:** Implement and test speed optimizations for Word2Vec (e.g., Negative Sampling/Hierarchical Softmax). |
| **Phase 5: Automation & Reliability** | **Workflow Orchestration and Monitoring** | 3 - 4 Weeks | **Job Scheduling:** Implement **Airflow/Prefect** workflow to manage the scheduled, periodic execution of the **Batch Re-clustering** (Step 5).<br>**Containerization:** Dockerize the entire application for reliable deployment in a cloud environment (e.g., Kubernetes/ECS).<br>**Inference Service:** Deploy the Step 4 classification logic as a low-latency **microservice** accessible via API.<br>**Logging & Alerting:** Set up comprehensive logging and implement basic health checks and alerts (e.g., if the outlier rate spikes). |
| **Phase 6: Advanced Tuning & Robustness** | **Model Drift and Edge Case Handling** | 1 - 2 Weeks | **Advanced Hyperparameter Search:** Use GridSearchCV/Bayesian Optimization to find the optimal $\epsilon$ and $\mathbf{min\_samples}$ across the full 10k dataset.<br>**Drift Detection:** Implement metrics to monitor the stability of the **Anomaly Threshold ($\Theta$)** and automatically alert if significant model drift is detected.<br>**Final Documentation:** Complete detailed runbooks and maintenance guides. |

## Total Estimated Duration for Future Work: 9 to 12 Weeks

This duration emphasizes that building a robust system that can reliably process and maintain its intelligence over a continuous data stream is a multi-month engineering effort, even after the core ML algorithm is proven.

# Attachment: Detailed Solution of the Project with Code Explanation

## File download

Assumption: The file download process does not need to be automatic or is already automatic

## Read the files

Assumption: Reading the file by a.exe is a process that does not need to be automatic or is already automatic

## Read .txt file

### Read the files

```
import os
file_contents = {}
for i in range(1, 100):
    file_name = f"{i}.txt"
    with open(file_name, 'r', encoding='utf-8') as f:
        file_contents[file_name] = f.read()
```

### Pre process

We might need to preprocess the files like remove excess whitespace (spaces, line breaks) and break down the code into meaningful Tokens like `opcode`, `instruction`, and `register`.

## Extract text dataset

There are python libraries to do this. For example, `scikit-learn`, `gensim`, `transformers` can do it.
It is the key point for converting the document to vector. And since these files are not in natural languages like English and Chinese, we will need to use some computer language specific models to solve the problem. Models like Sentence-Tranformer, general BERT is not applicable here.

### 1) Statistical and Structural Features

| Scheme | Technology/Model | Description | Applicable Scenarios |
|---|---|---|---|
| **Basic Statistics** | TF-IDF (Term Frequency-Inverse Document Frequency) | Treats **opcodes** or **N-gram sequences** (e.g., MOV EAX) as "vocabulary" and calculates their importance. Generates high-dimensional sparse vectors. | Suitable for distinguishing files based on **differences in common operations or commands**. **Low computational resource requirements.** |
| **Structural Statistics** | Feature Counting | Extracts non-textual features: counts structural metrics like the **number of functions**, **number of loops**, **specific dangerous API calls**, and **string lengths** within the file. | Suitable for distinguishing files with **different functional complexity or security focuses**. |
| **LSI/PCA** | Dimensionality Reduction | Reduces the dimensionality of TF-IDF vectors to decrease noise and improve clustering efficiency. | Optimizes high-dimensional sparse vectors to improve the performance of clustering algorithms (e.g., K-Means). |

## 2) Domain-Specific Deep Learning Embeddings

| Scheme | Technology/Model | Description | Applicable Scenarios |
|---|---|---|---|
| **Sequence Embedding** | Word2Vec / FastText (Skip-Gram/CBOW) | Treats opcodes in the file as "vocabulary" and trains vector representations of these "words" **internally within your file dataset**. | Captures the **co-occurrence relationships of opcodes/instructions**. Solves the **OOV problem** because the vocabulary is derived entirely from your data. |
| **Document Vector** | Averaging Word Vectors | Averages or weighted-averages all Word2Vec word vectors within a file to generate a **Document Embedding** (File Vector). | Represents each file as a fixed-length, semantically rich vector, suitable for clustering. |
| **Specialized Model** | CodeBERT / Unix-BERT | If a model is pre-trained on similar code (e.g., C/Shell), these models can be used to extract contextual vectors. | Suitable for more complex code semantic analysis, but **may require additional fine-tuning for assembly language**. |

These models are all offline models. Considering the real constrains of working environment, using online models (like OpenAI APIs) might not be allowed.

# Choice

To create a concrete, runnable plan, we must make a definitive choice. For the specific scenario—clustering files containing non-natural language (like kernel shell/assembly) where the goal is to detect structural or functional groups—I will select the `Word2Vec/FastText` approach, followed by `Averaging Word Vectors` to create the final document embedding. This method offers a robust balance of capturing symbolic relationships and generating clean, fixed-length vectors suitable for clustering algorithms like DBSCAN.

| Rationale | Explanation |
|---|---|
| Semantic Capture | Unlike TF-IDF, Word2Vec/FastText captures the co-occurrence context of opcodes (e.g., how often JMP follows CMP). This pattern is often indicative of functional similarity in code. |
| OOV & Vocabulary | Since it's trained on your specific dataset, it handles unique instruction sets and symbols (OOV problem) better than models pre-trained on natural language (like BERT/Sentence-Transformer). |
| Fixed-Length Vector | Averaging the word vectors produces a fixed-length vector for every file, which is a perfect input format for distance-based clustering algorithms like DBSCAN. |

## Execution Detail

**Tool:** Python's `gensim` library for Word2Vec/FastText.

1. **Tokenization:** Each file must first be tokenized.
    - **Input:** File Content (e.g., `MOV EAX, 10\nCALL FUNC_A`)
    - **Output:** List of Tokens (e.g., `['MOV', 'EAX', '10', 'CALL', 'FUNC_A']`)
    - **Action:** Implement a custom parser to separate opcodes, registers, and constants, while stripping comments and directives.

    We use `os` for file handling and define a custom function to clean comments and extract meaningful opcodes and symbols from structured code files.

```
import os
import re
from gensim.models import Word2Vec
import numpy as np

# Assuming files are located in the 'code_files' folder within the current directory
FILE_DIR = 'code_files'
NUM_FILES = 100
EMBEDDING_DIM = 100 # The chosen vector dimension

def custom_tokenizer(file_content: str) -> list:
    """
    Custom tokenization function:
    1. Removes inline comments (e.g., those starting with ';' or '#')
    2. Converts content to uppercase (optional, but helps normalize opcodes)
    3. Splits content using spaces, commas, etc., as delimiters
    4. Removes empty strings
    """
    tokens = []
```

```
    # 1. Process line by line and remove comments (adjust comment markers as needed)
    for line in file_content.split('\n'):
        # Remove inline comments
        if ';' in line:
            line = line[:line.index(';')]
        if '#' in line:
            line = line[:line.index('#')]

        # 2. Clean up whitespace and special characters, then split
        # Replace commas with spaces for unified handling
        line = line.replace(',', ' ').strip()

        # 3. Split by space and normalize to uppercase
        line_tokens = [token.upper() for token in line.split() if token]
        tokens.extend(line_tokens)

    return tokens

# Collection of tokens lists from all files (the Word2Vec training corpus)
corpus_tokens = []
file_names = []

print(f"1. Reading and tokenizing {NUM_FILES} files...")
for i in range(1, NUM_FILES + 1):
    file_name = f"{i}.txt"
    try:
        # Assuming file path is 'code_files/1.txt'
        file_path = os.path.join(FILE_DIR, file_name)
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()
            tokens = custom_tokenizer(content)

            if tokens:
                corpus_tokens.append(tokens)
                file_names.append(file_name)

    except FileNotFoundError:
        print(f"Warning: File {file_name} not found.")

if not corpus_tokens:
    raise ValueError("Corpus is empty. Please check file paths and content.")

print(f"Tokenization complete. Collected {len(corpus_tokens)} documents.")
```

2. **Word2Vec Training:** Train the embedding model using the tokens from files $1$ to $100$.
    - The model learns a vector (e.g., 100 dimensions) for every unique opcode/symbol in corpus.

Use the `gensim` library to train the Word2Vec model on custom code corpus.

```python
print("2. Training Word2Vec model...")

# Initialize and train the Word2Vec model
# vector_size: embedding dimension
# window: context window size
# min_count: ignores vocabulary words with a frequency less than this
model = Word2Vec(
    sentences=corpus_tokens,
    vector_size=EMBEDDING_DIM,
    window=5,
    min_count=1,
    workers=4 # Use multiple cores for speedup
)

# Lock the model to stop further training, improving memory efficiency
model.init_sims(replace=True)

print(f"Word2Vec model training complete. Vocabulary size:
{len(model.wv.key_to_index)}.")

# Example test: View the vector for the 'MOV' opcode
# print(f"\n'MOV' vector example (first 5 dimensions): {model.wv['MOV'][:5]}")
```

3. **Document Embedding:** Generate the final vector for each document.
    - For file $i$, take the average of all Word2Vec vectors of its tokens.
    - **Result:** A matrix $X$ of shape $(100, 100)$, where $100$ is the number of files and $100$ is the chosen embedding dimension.
      Calculate the average of the token vectors for each file to generate the final feature matrix $X$.

```python
def generate_document_vector(tokens: list, model: Word2Vec) -> np.ndarray:
    """
    Generates a document vector by averaging the Word2Vec vectors of all tokens in the
file.
    """
    valid_vectors = []

    # Iterate through each token in the file
    for token in tokens:
        # Check if the token is in the Word2Vec model's vocabulary
        if token in model.wv:
            valid_vectors.append(model.wv[token])

    if valid_vectors:
        # Return the average of all valid vectors
        return np.mean(valid_vectors, axis=0)
    else:
```

```
        # If the file is empty or has no known vocabulary, return a zero vector
        return np.zeros(model.vector_size)

print("3. Generating the final document feature matrix X...")

# Initialize the feature matrix X
# Matrix shape is (number of files, vector dimension)
X = np.zeros((len(corpus_tokens), EMBEDDING_DIM))

# Iterate through all documents, generate vectors, and populate matrix X
for i, tokens in enumerate(corpus_tokens):
    doc_vector = generate_document_vector(tokens, model)
    X[i] = doc_vector

print(f"Feature matrix X generation complete. Shape: {X.shape}")

# X is now the input data for the subsequent DBSCAN clustering.
# Each row of the matrix represents the 100-dimensional feature vector for one file.
```

# Model Training & Initial Clustering

**Tool:** Python's `scikit-learn` or `hdbscan` library.

1. **DBSCAN Application:** Apply DBSCAN directly to the feature matrix $X$.
    - **Output:** A list of cluster labels (e.g., $0, 1, 2, \ldots$), where $-\mathbf{1}$ specifically denotes the initial **outliers**.

```
from sklearn.cluster import DBSCAN
import numpy as np
# from the previous step: X is the (N, EMBEDDING_DIM) feature matrix

# --- Configuration for DBSCAN ---
# These parameters are crucial and often require tuning based on given data.
# eps (epsilon): The maximum distance between two samples for one to be considered as
in the neighborhood of the other.
# min_samples: The number of samples (or total weight) in a neighborhood for a point to
be considered as a core point.
EPSILON = 0.5  # A starting value; adjust based on the density of code embeddings
MIN_SAMPLES = 5 # A reasonable minimum number of files to form a valid group

print("1. Applying DBSCAN clustering to the feature matrix X...")

# Initialize and fit the DBSCAN model
db = DBSCAN(eps=EPSILON, min_samples=MIN_SAMPLES, metric='cosine').fit(X)

# Retrieve the cluster labels for each file
labels = db.labels_
```

```python
# Determine the number of clusters found (excluding noise/outliers)
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_outliers_ = list(labels).count(-1)

print(f"DBSCAN clustering complete.")
print(f"  - Number of estimated clusters (groups): {n_clusters_}")
print(f"  - Number of estimated outliers (label -1): {n_outliers_}")

# Example output of labels (e.g., [0, 0, 1, -1, 0, 1, ...])
# print("\nFirst 10 cluster labels:", labels[:10])
```

2. **Cluster Center Calculation:** Calculate the **centroid** (mean vector) for each group $C_k$ (where $k \neq -1$). These centroids define the core identity of each group.

```python
# Initialize dictionaries to store centroids and group indices
cluster_centroids = {}
unique_labels = set(labels)

print("\n2. Calculating Centroids for each defined cluster...")

for k in unique_labels:
    if k == -1:
        # Skip the outlier/noise points
        continue

    # Get the indices of the data points belonging to cluster k
    class_member_mask = (labels == k)

    # Extract the feature vectors for the current cluster
    cluster_points = X[class_member_mask]

    # Calculate the centroid (mean vector) for this cluster
    centroid = np.mean(cluster_points, axis=0)

    # Store the centroid
    cluster_centroids[k] = centroid

    print(f"  - Centroid for Group {k} calculated (Size: {len(cluster_points)} files)")

# Example: Access the centroid of Group 0
# centroid_group_0 = cluster_centroids.get(0)
# print(f"\nCentroid of Group 0 (first 5 dimensions): {centroid_group_0[:5]}")

# cluster_centroids now holds the core identity vectors (C_k) for all valid groups.
# This structure is essential for the next step: classifying new files and checking
deviation.
```

# New File Classification & Incremental Model Update

This step uses the established groups to handle the new file $101.txt$.

1. **New File Encoding:** Tokenize $101.txt$ and use the **already trained Word2Vec model** from Step 2 to generate its embedding vector $V_{101}$.
2. **Distance and Threshold Check:**
   - Calculate the **minimum distance** $D_{min}$ from $V_{101}$ to all existing cluster centroids $C_k$.
   - Compare $D_{min}$ against a pre-determined **threshold** $\Theta$ (e.g., $\mu + 2\sigma$ derived from the original cluster distances).

```python
# Assume X, labels, and cluster_centroids are available from Step 3.
# Also assume the necessary functions (calculate_threshold, Z_FACTOR) are defined.

def calculate_threshold(X: np.ndarray, labels: np.ndarray, centroids: Dict[int,
np.ndarray], z_factor: float = 2.0) -> float:
    # Function body from the previous response (calculates μ + Zσ)
    intra_cluster_distances = []
    for k in centroids.keys():
        distances = [cosine(point, centroids[k]) for point in X[labels == k]]
        intra_cluster_distances.extend(distances)
    if not intra_cluster_distances: return float('inf')
    mu, sigma = np.mean(intra_cluster_distances), np.std(intra_cluster_distances)
    return mu + z_factor * sigma

# --- Execute Step 2a: Calculate Threshold (Theta) ---
Z_FACTOR = 2.0
THRESHOLD = calculate_threshold(X, labels, cluster_centroids, Z_FACTOR)
print("\n2. Distance and Threshold Check:")
print(f"   2a. Anomaly Threshold (Theta, μ + {Z_FACTOR}σ): {THRESHOLD:.4f}")

# --- Execute Step 2b: Calculate Minimum Distance (D_min) ---
min_distance = float('inf')
closest_cluster = -1

for k, centroid in cluster_centroids.items():
    distance = cosine(V_new, centroid)

    if distance < min_distance:
        min_distance = distance
        closest_cluster = k

print(f"   2b. Minimum Distance (D_min) to closest centroid: {min_distance:.4f}")
```

3. **Decision:**
   - If $D_{min} \le \Theta$: Assign $101.txt$ to the closest group.

- If $D_{min} > \Theta$: **Identify $101.txt$ as a new group/outlier.** This triggers the need to revise the model (the **periodic re-clustering** mechanism).

```
# Assume min_distance, closest_cluster, and THRESHOLD are available from Step 2.

print("\n3. Decision:")

if min_distance <= THRESHOLD:
    # D_min is within the acceptable range.
    final_decision = f"ASSIGNED_TO_GROUP"
    print(f"   Decision: ASSIGN to existing Group {closest_cluster}")

elif min_distance > THRESHOLD:
    # D_min is outside the acceptable range (Excessive Deviation).
    final_decision = "EXCESSIVE_DEVIATION_NEW_GROUP"
    print(f"   Decision: FLAG as OUTLIER/NEW GROUP")
    print(f"   ACTION: Trigger periodic batch re-clustering.")

# Example structure for the result
result_dict = {
    "file": "101.txt",
    "D_min": min_distance,
    "Threshold": THRESHOLD,
    "Decision": final_decision,
    "Assigned_Group": closest_cluster
}
# print("\nFinal Result Summary:", result_dict)
```

# Reclustering

The reclustering step provides the mechanism for model evolution and addresses the limitations of the initial DBSCAN run. As we get more and more files from input, there will definitely new files that are not part of the original groups. In this cases, outliers will become more and more so we need to recluster.

1. **Data Preparation and Full Retraining:** This section loads the expanded dataset and re-runs both the Word2Vec model training and the document embedding generation.

```
# Assume functions and constants from Step 2 (e.g., custom_tokenizer, EMBEDDING_DIM)
are available.

def load_all_data_for_retraining():
    """
    Simulates loading the original 100 files AND all files from the Outlier Queue.
    Returns: (list of file tokens, list of raw file contents)
    """
    # Placeholder: In a real system, this would fetch files from a database/storage.
    # For this example, we assume we get a new, larger list of tokens:
```

```
    # Simulating 100 initial files + 10 new outliers = 110 files
    return [
        ['MOV', 'EAX', '10'],
        ['PUSH', 'EBP'],
        # ... up to 110 documents ...
        ['NEW_OP', 'XYZ']
    ]

def perform_full_retraining():

    print("\n--- Starting Model Retraining (Full Batch) ---")

    # 1. Load the entire expanded corpus
    new_corpus_tokens = load_all_data_for_retraining()

    if not new_corpus_tokens:
        print("Error: Expanded corpus is empty.")
        return None, None, None

    # 2. Re-train the Word2Vec model on the larger corpus
    # This captures new vocabulary and updates all existing vectors
    new_model = Word2Vec(
        sentences=new_corpus_tokens,
        vector_size=EMBEDDING_DIM,
        window=5,
        min_count=1,
        workers=4
    )
    new_model.init_sims(replace=True)
    print(f"Word2Vec Retrained. New Vocabulary Size: {len(new_model.wv.key_to_index)}")

    # 3. Re-generate the full feature matrix X'
    X_prime = np.array([
        generate_document_vector(tokens, new_model)
        for tokens in new_corpus_tokens
    ])

    print(f"New Feature Matrix X' Generated. Shape: {X_prime.shape}")
    return new_model, X_prime, new_corpus_tokens
```

This detailed plan now ensures continuity from feature selection through the final classification goal.