i want to develop a AI/ML WAF module.it should detect pre existing attacks and zero-day attacks as well. it should have a feedback system as well to train accordingly to user feedback. so as a starting step i downloaded ATRDF 2023 dataset. now guide me each step what to do and give me full codes of each program. iam new to AI/ML. i have base in security but not AI/ML.

You can absolutely build this, but it is a multi-step project and too large for a single "full code" drop in one message, especially since you are new to ML. The best way is to go in stages: first learn to train and evaluate a basic model on ATRDF, then later wrap it as a WAF with feedback and gradually add zero-day detection.[1] [2]

Below is a concrete, beginner-friendly roadmap and first code pieces. If this direction looks good, the next messages can go deeper into each file (training script, serving API, feedback loop, etc.).

Overall architecture

For a first version, aim for this:

- Offline:
  - Load ATRDF 2023 HTTP logs.
  - Extract text features from requests (URL, query, body, headers).[2] [1]
  - Train a classifier that:
    - Model 1: Binary (benign vs attack).
    - Model 2: Multi-class (attack types like SQLi, XSS, RCE, etc.).[1] [2]
  - Save the trained model and vectorizer to disk.
- Online WAF (Python service in front of your API):
  - Accept an incoming HTTP request.
  - Convert it to the same text representation as in training.
  - Run through vectorizer + model.
  - Decide allow/block and log the decision.

- Feedback and zero-day:
  - Store each decision plus “user feedback” (e.g., operator marks a request as wrongly classified).
  - Periodically retrain / fine-tune the model using this new labeled data.
  - For zero-day, later add:
    - An anomaly detector (e.g., IsolationForest) on top of benign traffic.
    - Or a more advanced model (BERT-style request encoder) based on what is being explored on ATRDF.[3] [4] [2]

Below is step 1 with full, runnable example code: basic training + evaluation on ATRDF-style JSON using scikit-learn.

Step 1: Prepare environment and understand ATRDF

From the challenge description, ATRDF provides:[2] [1]

- JSON lines / lists of HTTP requests and responses.
- A label for each request: benign or specific attack type (SQLi, XSS, RCE, Directory Traversal, Cookie Injection, Log4j, etc.).[1] [2]

Do this first:

1. Install Python packages (in a virtualenv is best):

- pip install:
  - pandas
  - scikit-learn
  - joblib
  - numpy

2. Inspect the dataset structure:

- Open one training JSON file from ATRDF and see:
  - Field names for:
    - URL/path
    - HTTP method
    - Query string / parameters
    - Body
    - Headers
  - Label field name for attack type (e.g. "label", "attack_type", or similar) as described in the challenge.[1]

Because file structure may differ slightly, the code below assumes:

- Each dataset file is JSON lines (one JSON object per line), with fields:

- "method"
- "path"
- "query"
- "body"
- "headers"
- "label" (string label: "Benign", "SQLi", "XSS", etc.)

If the field names differ, you'll just update a small helper function.

Step 2: Basic training code (single script)

Create a file train_waf_model.py with the following code. This:

- Loads a JSONL file.
- Builds a text representation of each request.
- Uses a character-level TF-IDF vectorizer (good baseline for payload detection).[5] [6]
- Trains:
  - A binary classifier.
  - A multi-class classifier.
- Saves trained models and vectorizers.

(You only need one to start; here you get both to see the difference.)

```
# train_waf_model.py

import json
import os
from typing import List, Tuple

import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import LabelBinarizer
from joblib import dump


def load_atrdf_jsonl(path: str) -> pd.DataFrame:
    """
    Load ATRDF dataset from a JSON lines file.
    Adjust field names if your JSON structure is different.
    """
    records = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
```

```
                continue
            obj = json.loads(line)
            records.append(obj)

    df = pd.DataFrame(records)
    return df


def build_request_text(row: pd.Series) -> str:
    """
    Build a single string that represents the HTTP request.
    Modify this based on the exact ATRDF fields.
    """
    method = str(row.get("method", ""))
    path = str(row.get("path", ""))
    query = str(row.get("query", ""))
    body = str(row.get("body", ""))
    headers = row.get("headers", {})

    # Convert headers dict to simple text
    if isinstance(headers, dict):
        headers_text = " ".join(f"{k}:{v}" for k, v in headers.items())
    else:
        headers_text = str(headers)

    # Concatenate everything into one text
    # You can try different formats; simple concat works fine for a start.
    text = f"{method} {path} {query} {body} {headers_text}"
    return text


def prepare_dataset(df: pd.DataFrame) -> Tuple[List[str], List[str]]:
    """
    Convert DataFrame rows into:
    - texts: list of strings
    - labels: list of attack labels
    """
    # Confirm the label column name by inspecting df.columns
    if "label" not in df.columns:
        raise ValueError(f"'label' column not found. Available columns: {df.columns}")

    texts = df.apply(build_request_text, axis=1).tolist()
    labels = df["label"].astype(str).tolist()
    return texts, labels


def train_binary_classifier(texts: List[str], labels: List[str]):
    """
    Train a binary classifier: benign vs attack.
    Assumes label 'Benign' (exact string) is used for normal traffic.
    Adjust the benign label string as needed.
    """
    # Convert to binary labels
    benign_label_value = "Benign"  # change if different in your dataset
    y_binary = np.array([1 if lbl == benign_label_value else 0 for lbl in labels])
```

```
    # Vectorizer: character-level TF-IDF
    vectorizer = TfidfVectorizer(
        analyzer="char",
        ngram_range=(3, 5),  # 3-5 char n-grams is common for web payloads
        min_df=2
    )

    X = vectorizer.fit_transform(texts)

    X_train, X_test, y_train, y_test = train_test_split(
        X, y_binary, test_size=0.2, random_state=42, stratify=y_binary
    )

    clf = LogisticRegression(
        max_iter=500,
        n_jobs=-1
    )
    clf.fit(X_train, y_train)

    # Evaluation
    y_pred = clf.predict(X_test)
    print("=== Binary classifier report (Benign vs Attack) ===")
    print(classification_report(y_test, y_pred))
    print("Confusion matrix:")
    print(confusion_matrix(y_test, y_pred))

    return vectorizer, clf


def train_multiclass_classifier(texts: List[str], labels: List[str]):
    """
    Train a multi-class classifier on specific attack types.
    """
    vectorizer = TfidfVectorizer(
        analyzer="char",
        ngram_range=(3, 5),
        min_df=2
    )
    X = vectorizer.fit_transform(texts)

    X_train, X_test, y_train, y_test = train_test_split(
        X, labels, test_size=0.2, random_state=42, stratify=labels
    )

    clf = LogisticRegression(
        max_iter=1000,
        n_jobs=-1
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)
    print("=== Multi-class classifier report (attack types) ===")
    print(classification_report(y_test, y_pred))
    print("Confusion matrix:")
    print(confusion_matrix(y_test, y_pred))
```

```
    return vectorizer, clf


def main():
    # TODO: change to your actual path
    dataset_path = "dataset_1_train.jsonl"

    print(f"Loading data from {dataset_path} ...")
    df = load_atrdf_jsonl(dataset_path)
    print("Columns:", df.columns)

    texts, labels = prepare_dataset(df)
    print(f"Loaded {len(texts)} samples.")

    # Train binary classifier
    bin_vectorizer, bin_clf = train_binary_classifier(texts, labels)

    # Train multi-class classifier
    mc_vectorizer, mc_clf = train_multiclass_classifier(texts, labels)

    # Save models
    os.makedirs("models", exist_ok=True)
    dump(bin_vectorizer, "models/binary_vectorizer.joblib")
    dump(bin_clf, "models/binary_model.joblib")
    dump(mc_vectorizer, "models/mc_vectorizer.joblib")
    dump(mc_clf, "models/mc_model.joblib")

    print("Models saved in 'models' directory.")


if __name__ == "__main__":
    main()
```

Run it with:

- python train_waf_model.py

Once this works and you see reasonable classification metrics (precision/recall ~ high 90s is achievable on ATRDF with good models).[4] [3] [2]

Step 3: Simple prediction API (WAF core)

Next step is to expose this model as a simple HTTP service that your real WAF / reverse proxy can call. For a start, use Flask or FastAPI.

Example: create waf_api.py:

```
# waf_api.py

import json
from typing import Dict, Any

from fastapi import FastAPI, Request
from pydantic import BaseModel
from joblib import load
```

```
from train_waf_model import build_request_text  # reuse same logic

app = FastAPI()

# Load models
bin_vectorizer = load("models/binary_vectorizer.joblib")
bin_model = load("models/binary_model.joblib")

mc_vectorizer = load("models/mc_vectorizer.joblib")
mc_model = load("models/mc_model.joblib")


class WAFRequest(BaseModel):
    method: str
    path: str
    query: str = ""
    body: str = ""
    headers: Dict[str, Any] = {}


@app.post("/classify")
async def classify(req: WAFRequest):
    # Convert request to text
    row = {
        "method": req.method,
        "path": req.path,
        "query": req.query,
        "body": req.body,
        "headers": req.headers
    }
    text = build_request_text(row)

    # Binary prediction
    X_bin = bin_vectorizer.transform([text])
    bin_pred = bin_model.predict(X_bin)[^1_0]
    bin_label = "Benign" if bin_pred == 1 else "Attack"

    # Multi-class prediction
    X_mc = mc_vectorizer.transform([text])
    mc_pred = mc_model.predict(X_mc)[^1_0]

    # Return decision; in a real WAF you would also log this
    return {
        "binary_label": bin_label,
        "attack_type": mc_pred
    }
```

Run with:

- pip install fastapi uvicorn
- uvicorn waf_api:app --reload --port 8000

Then send JSON requests (e.g. with curl, Postman, or from your reverse proxy) with method/path/query/body/headers, and this service will respond with the decision.

Step 4: Add simple feedback storage

Design a very simple feedback schema:

- For each request you classified:
  - Store:
    - request_text
    - predicted_label / attack_type
    - operator_feedback: “correct” or corrected_label

At first, just store feedback in a CSV:

- Columns:
  - text, original_label, predicted_label, corrected_label

Later, have a retraining script that:

- Loads original ATRDF data.
- Adds feedback rows as extra samples with corrected_label.
- Retrains and overwrites the models.

Example feedback receiver added to the same waf_api.py:

```
# Add under the imports in waf_api.py
import csv
from pathlib import Path

FEEDBACK_FILE = Path("feedback.csv")

# Ensure file has header
if not FEEDBACK_FILE.exists():
    with FEEDBACK_FILE.open("w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["text", "predicted_binary", "predicted_attack_type", "correct_la


class FeedbackItem(BaseModel):
    text: str  # same text used for classification
    predicted_binary: str
    predicted_attack_type: str
    correct_label: str  # what the operator says is correct


@app.post("/feedback")
async def feedback(item: FeedbackItem):
    with FEEDBACK_FILE.open("a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            item.text,
            item.predicted_binary,
            item.predicted_attack_type,
            item.correct_label
```

```
        ])
    return {"status": "ok"}
```

Then you:

1. When you classify a request, include the text and predictions in the UI/log.
2. Operator sends a POST to /feedback with the corrected label.

In a follow-up script (e.g., retrain_with_feedback.py), you can:

- Read feedback.csv.
- Treat each row as a labeled sample (text, correct_label).
- Append to your ATRDF training set before training.

Step 5: Adding anomaly detection for zero-day

Classic classifiers detect known patterns well but struggle with zero-day. Once you are comfortable with the above:[7] [6]

- Train a "normal traffic only" model (one-class anomaly detection):
  - Take only benign requests from ATRDF.
  - Use TF-IDF again, but train a OneClassSVM or IsolationForest.
  - During serving:
    - If binary classifier says benign but anomaly detector says "outlier", treat as suspicious and maybe block or send to manual review.
- There are recent papers that use Transformers and contrastive learning on web requests for better unknown attack detection, also evaluated on ATRDF. Those can be your next stage, but they are more complex than scikit-learn.[3] [4] [2]

Next step

Because this is a big project and you are new to ML, the very next concrete step:

- Tell the exact format of one ATRDF sample you have (paste one JSON object with keys, but remove any sensitive info).
- Then the code can be adjusted precisely to your real field names.
- After that, the next messages can include:
  - retrain_with_feedback.py (to actually use feedback for training),
  - a simple reverse proxy example (e.g. using Python + httpx) that calls waf_api,
  - and a basic strategy to evaluate zero-day behaviour.

⁂

1. https://github.com/ArielCyber/Cisco_Ariel_Uni_API_security_challenge
2. https://arxiv.org/html/2405.11258v1

3. https://www.sciencedirect.com/science/article/abs/pii/S0167404824005558
4. https://dl.acm.org/doi/10.1016/j.cose.2024.104249
5. https://github.com/kiannaquines/web-application-firewall-machine-learning
6. https://onlinelibrary.wiley.com/doi/10.1155/2022/5280158
7. https://downloads.hindawi.com/journals/js/2023/6048087.pdf
8. https://www.mdpi.com/1999-5903/15/7/231
9. https://dl.acm.org/doi/10.1145/3630253
10. http://www.emerald.com/ijwis/article/21/1/77-95/1239795
11. https://www.mdpi.com/2078-2489/15/4/199
12. https://jtiik.ub.ac.id/index.php/jtiik/article/view/8911
13. https://www.tandfonline.com/doi/full/10.1080/1206212X.2024.2448835
14. https://ieeexplore.ieee.org/document/10972011/
15. https://ieeexplore.ieee.org/document/10432841/
16. https://arxiv.org/pdf/2502.03909.pdf
17. https://www.mdpi.com/1424-8220/23/13/5941/pdf?version=1687924880
18. https://pmc.ncbi.nlm.nih.gov/articles/PMC9720441/
19. http://arxiv.org/pdf/2407.11278.pdf
20. http://arxiv.org/pdf/2404.07765.pdf
21. https://arxiv.org/pdf/2308.12627.pdf
22. https://linkinghub.elsevier.com/retrieve/pii/S2352340923009605
23. https://arxiv.org/html/2405.11247v1
24. https://ieeexplore.ieee.org/iel8/11036321/11036274/11036344.pdf
25. https://github.com/abhisheks008/ML-Crate/issues/502
26. https://pmc.ncbi.nlm.nih.gov/articles/PMC12453791/
27. https://gist.github.com/stefanbschneider/96602bb3c8b256b90058d59f337a0e59
28. https://papers.ssrn.com/sol3/Delivery.cfm/b1eec7d8-c9fd-400b-ab6e-70728a9311ea-MECA.pdf?abstractid=4927579&mirid=1
29. https://github.com/vladan-stojnic/ML-based-WAF
30. https://www.scribd.com/document/864296932/Few-Shot-API-Attack-Anomaly-Detection-in-a-Classification-by-Retrieval-Framework
31. https://dev.to/sapnilcsecu/step-by-step-web-application-firewall-waf-development-by-using-multinomial-native-bayes-algorithm-fdd
32. https://github.com/ArielCyber
33. https://www.youtube.com/watch?v=Nlg0OOG2QYA
34. https://sist.sathyabama.ac.in/sist_naac/documents/1.3.4/1822-b.e-cse-batchno-102.pdf
35. https://quokkalabs.com/blog/ai-web-application-firewall/
36. https://dl.acm.org/doi/10.1145/3658644.3670388
37. https://arxiv.org/abs/2305.19254