Below I give a deep analysis of the requirements (by page), then a practical, step-by-step solution & engineering procedure you can implement end-to-end (architecture, data pipeline, models, integration with an open-source WAF, GUI, tests, metrics, and final deliverables). I map each part to the scoring and deliverable items in the PDF so nothing is missed.

**1) Brief summary of the PDF requirements (key points)**

- Goal: build an ML module that baselines network/web traffic and performs anomaly detection to augment a WAF. (page 1).

Hackathon Challenge 3_WAF

- Must handle inbound/outbound HTTP(S) content, learning normal baselines and flagging deviations. Support supervised, unsupervised and semi-supervised models with explainability. (page 1).

Hackathon Challenge 3_WAF

- Convert ML insights into human-readable security rules administrators can approve/deploy. (page 2).

Hackathon Challenge 3_WAF

- Must be high-performance, low-latency; support encrypted traffic handling (TLS termination or decryption layer). Continuous learning & admin feedback loop required. (page 2).

Hackathon Challenge 3_WAF

- Evaluation scenarios include baseline traffic, encrypted traffic, zero-day attacks, API abuse & bot traffic. Scoring emphasizes detection accuracy, false positives, performance, explainability, and rule recommendation quality. (page 2–3).

Hackathon Challenge 3_WAF

- Final deliverables: working ML module + dashboard, source code + README, 5-minute demo video, 2–3 page tech doc, logs/metrics, an 8–10 slide presentation. (page 3–4).

Hackathon Challenge 3_WAF

**2) High-level architecture (recommended)**

Goal: balance *real-time blocking* (low latency) and *deep analysis* (high accuracy). Use hybrid pipeline: lightweight inline scoring + asynchronous deep analysis + human approval for rule deployment.

Components:

1. **Traffic Collector / Ingress**
   - Deployed at TLS termination point (reverse proxy / load balancer) so HTTPS can be analyzed after decryption. Options: terminate TLS at Envoy / NGINX or use a sidecar TLS terminator. (This addresses encrypted traffic requirement).
2. **Preprocessor & Feature Extractor (real-time)**
   - Extract fast, deterministic features (URL length, header anomalies, char counts, parameter entropy, token TF-IDF hashed signatures, request rate features) in <1 ms per request. Written in Go/C++ or fast Python (C extensions).
3. **Lightweight Real-time Model (inline)**
   - A fast model (e.g., XGBoost or small neural net exported to ONNX/Treelite) that returns a risk score. Low latency, used for monitor/challenge/block decisions.
4. **Async Deep Analysis Service**
   - More expensive models: Transformer / CNN on tokenized bodies, autoencoders for anomalies, graph-based behavioral models (sessions/IP graphs). Run asynchronously; results feed back to learning & rule suggestion engine.
5. **Rule Recommendation Engine**
   - Transliterate ML findings into candidate ModSecurity or NGINX rules (human readable templates), with confidence & explanation attached.
6. **Decision Engine & Policy Store**
   - Combines rule logic + ML risk score + admin policy. Exposes API for WAF integration (ModSecurity, NGINX, or an edge proxy).
7. **GUI / Dashboard + Feedback Loop**
   - Dashboard for alerts, timelines, top features, SHAP explanations, approve/reject rule suggestions. Admin feedback is logged and used to retrain models.
8. **Data Lake / Feature Store**
   - Kafka → Parquet in object store (S3) or Elasticsearch for indexing. Feature store for model training.
9. **Monitoring / Perf**
   - Prometheus + Grafana for latency/throughput; logging for FP/FN tracking.

Diagram (conceptually): TLS Termination → Preprocessor → [Inline Model → Decision Engine → WAF action] AND → [Async analyzer → Rule Recommender → Dashboard → Admin approval → Rule store → WAF]

**3) Data & features (what to capture and why)**

Capture per request:

- Metadata: timestamp, src IP (anonymize if required), destination, HTTP method, response code, response time.
- Request content: URL path, query string, headers, cookies, body (where allowed/sanitized), content-type.
- Session context: device fingerprint, rate (requests per IP/session), previous N requests for session-aware features.

Feature categories:

- Lexical: lengths, char counts, percent of non-ASCII, URL-encoded sequences count, base64 entropy.
- Syntactic: presence of SQL keywords, script tags, function calls, regex matches for common exploits.
- Statistical: param entropy, number of params, repeated params, value distributions.
- Behavioral: requests/sec per IP, session path patterns, unusual time-of-day.
- Derived: TF-IDF / hashed n-grams (char n-grams for obfuscation resilience), embeddings from a token model.
- Graph features: new IP→endpoint connections, degree centrality for IPs, cross-service calls.

Sanitization: remove PII before storing for training. Keep hashing/anonymization for IPs if required.

**4) Models & approaches (match to challenge: supervised / unsupervised / semi)**

- **Baseline (fast & explainable)**: XGBoost / LightGBM on engineered features. Good for initial rules & explainability (feature importances).
- **Unsupervised / Anomaly detection**: Isolation Forest, PCA/LOF, autoencoder (tabular or sequence autoencoder) to detect deviations from baseline. Good for zero-day.
- **Sequence models (deep)**: small Transformer or 1D-CNN over tokenized request bodies/URLs to capture obfuscations. Use char-level tokenization to catch encodings.

- **Session / Graph models**: Graph Neural Network or temporal models (LSTM/Transformer on request sequences per session/IP) to detect API abuse / bots.
- **Ensemble strategy**: inline XGBoost for quick decisions + async deep models for confirmation and rule creation.

Explainability:

- Use SHAP for tree models (fast, per-request local explanations). For neural sequence models, use attention visualizations and integrated gradients approximations. Always attach top-N contributing features/phrases to alerts.

**5) Rule recommendation (how to translate ML outputs to WAF rules)**

Procedure:

1. ML finds high-risk requests and identifies top contributing tokens/fields (e.g., parameter q contains UNION SELECT).
2. Map token/pattern to a templated ModSecurity rule:
   - Template: SecRule REQUEST_URI|REQUEST_BODY "@contains 'UNION SELECT'" "id:1001,phase:2,deny,log,msg:'ML sugg: SQLi fragment in param q (score=0.94)'"
3. Attach metadata: confidence score, examples, SHAP explanation, suggested action (monitor/challenge/block).
4. Present candidate rules in dashboard for admin approval. On approval, push to WAF rule repo and hot-reload (ModSecurity supports reloading or external rule include).
5. Keep rule provenance: link rule → ML event IDs → training examples to allow retraining / rollback.

To integrate with other WAFs, produce rules in canonical formats (ModSecurity, NGINX lua snippet, or custom API to the WAF).

**6) Encrypted traffic handling**

Options (explicitly required by the brief):

- **Preferred**: Terminate TLS at a reverse proxy (Envoy/NGINX) where the ML module is colocated or gets a copy (via sidecar) of decrypted traffic. This is the simplest for the hackathon. (page 2).

Hackathon Challenge 3_WAF

- **If TLS cannot be terminated**: use TLS inspection appliance or passive TLS fingerprinting (SNI, JA3, traffic metadata) and rely on behavioral features (session rates, endpoint access patterns) for anomaly detection. Document the limitations in the technical note. (page 2 scenarios).

Hackathon Challenge 3_WAF

**7) Performance & low-latency strategy**

- Inline model: keep features simple and model size tiny. Pre-compile models (Treelite for XGBoost, ONNX runtime for neural nets).
- Use asynchronous heavy analysis so inline path stays ≤ target latency (for demo, aim <10–20ms inference).
- Batch processing for heavy models; use GPU for expensive model training/inference (not necessary for inline).
- Use efficient I/O (gRPC) between WAF and ML service; instrument and test throughput with realistic load (e.g., 10k req/s) and measure P99 latency.

**8) Continuous learning & feedback loop**

- Dashboard allows label feedback (true/false positive) which is stored in the dataset.
- Periodic retraining pipeline (weekly or triggered by drift). Use automated training: data snapshot → feature calc → train/validate → promote model if performance improves.
- Drift detection: monitor feature distributions (KS tests) and model score distribution. Trigger retrain when drift threshold exceeded. (page 2: continuous learning).

Hackathon Challenge 3_WAF

**9) Tests & scenario simulations (mapped to PDF scenarios)**

Create test harnesses to generate data for each evaluation case:

1. **Baseline traffic scenario**
    - Scripted normal traffic (web crawls, API call patterns) + injected benign irregularities. Evaluate false positive rate. (page 2–3).

Hackathon Challenge 3_WAF

2. **Encrypted traffic handling**
    - Deploy TLS termination in testbed; feed HTTPS traffic. Also test passive metadata approach and document differences. (page 2).

Hackathon Challenge 3_WAF

3. **Zero-day attack simulation**

    - Create mutated attack payloads (obfuscated SQLi/XSS), use fuzzers and payload mutation to test detection of previously unseen variants. Evaluate anomaly detector recall vs signature detectors. (page 2–3).

Hackathon Challenge 3_WAF

4. **API abuse & bot traffic**
    - Simulate credential stuffing, scraping (fast repetitive requests from many IPs or rotating proxies), and slow high-latency bots. Use session/graph features to detect. (page 2).

Hackathon Challenge 3_WAF

Validation metrics:

- Primary: Detection Recall on attack set (prioritize this in scoring).
- Operational: False Positive Rate (keep low).
- Performance: throughput and P95/P99 latency.
- Explainability: qualitative scoring (are explanations useful to admins?). (these match the evaluation criteria in PDF).

Hackathon Challenge 3_WAF

**10) Implementation plan — step-by-step with milestones**

I give a practical 8-phase plan you can implement in a hackathon or short project:

**Phase 0 — Preparation (Day 0–1)**

- Choose tech stack: WAF base (ModSecurity + OWASP CRS recommended), reverse proxy (NGINX/Envoy), Kafka, Python (FastAPI), PyTorch/Scikit-learn/XGBoost, Elasticsearch or PostgreSQL, Prometheus/Grafana, Docker.
- Setup repo & CI.

**Phase 1 — Data & Ingest (Day 1–3)**

- Build simple traffic generator and log collector (nginx + access logs or capture pcap & parse).
- Implement parser to normalize requests into JSON events: {ts, ip, method, uri, headers, body, resp_code}.
- Store to Kafka and index a small dataset (for initial training).

**Phase 2 — Feature extraction + Baseline model (Day 3–6)**

- Implement feature extractor for real-time features (fast functions). Unit tests for feature correctness.
- Train a baseline XGBoost model on synthetic labeled set (OWASP payloads as positives + generated normal traffic as negatives). Evaluate.

**Phase 3 — Inline inference + WAF integration (Day 6–8)**

- Expose model via a fast REST/gRPC endpoint or compile model to a shared library.
- Integrate with ModSecurity: on request, call ML service; add ModSecurity action to block/monitor based on score. For demo, it's OK to run WAF in detection mode and show the suggested block. (meets pass/fail gate: integration).

Hackathon Challenge 3_WAF

**Phase 4 — Async deep analyzer & rule recommender (Day 8–11)**

- Build async worker that consumes suspicious events and runs deeper models.
- Implement rule templating engine to generate ModSecurity rules from ML outputs. Save suggestions to DB for admin approval.

**Phase 5 — Dashboard & Explainability (Day 9–13)**

- Build simple GUI (React or Plotly Dash) that shows timeline, top alerts, SHAP explanation (for tree models), and rule suggestion approval flow. Ensure it's user friendly (pass gate: dashboard).

Hackathon Challenge 3_WAF

**Phase 6 — Simulations & Eval (Day 11–14)**

- Run the scenario tests (baseline, encrypted via TLS termination, zero-day fuzz, bot scripts). Collect metrics (TP/FP, latency). Create logs & reports required by final deliverables. (page 3).

Hackathon Challenge 3_WAF

**Phase 7 — Documentation & Demo artifacts (Day 13–15)**

- Produce: source code + README, 2–3 page technical doc, 5-min demo recording showing: live traffic, anomaly detection, rule suggestion + approval, dashboard, and metrics. Prepare 8–10 slides. (PDF deliverables mapping).

Hackathon Challenge 3_WAF

**11) Minimum viable implementation (if time is short)**

Deliver a functioning subset that satisfies pass/fail and scores well:

- Inline XGBoost model + feature extractor integrated with ModSecurity (detection mode) — real-time detection (pass).
- Dashboard showing alerts + SHAP explanations — user friendly (pass).
- Rule recommendation engine producing ModSecurity rule templates for admin approval (pass).
- Simulate encrypted traffic by terminating TLS at nginx and feeding HTTPS to the system (pass).
  This subset covers the core objectives and required deliverables; you can mention missing advanced parts (e.g., full transformer deep models) as future improvements in documentation (the PDF allows partial submissions and reasons for noncompliance).

Hackathon Challenge 3_WAF

**12) Concrete code snippet — tiny starter (feature extraction + XGBoost)**

Below is a short Python skeleton you can copy into your repo and expand. This is only a starting point for the baseline model.

```
# requirements: pandas, scikit-learn, xgboost, shap, fastapi

import re

import pandas as pd

import numpy as np

from xgboost import XGBClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report

import shap


def extract_features(req):

    # req: dict with 'uri','body','headers','method'

    s = (req.get('uri','') + ' ' + req.get('body','')).lower()

    features = {}

    features['uri_len'] = len(req.get('uri',''))

    features['body_len'] = len(req.get('body','') or '')
```

```
    features['num_params'] = s.count('=')  # naive
    features['num_select'] = int(bool(re.search(r'\bselect\b', s)))
    features['num_union'] = int(bool(re.search(r'\bunion\b', s)))
    features['num_script'] = int(bool(re.search(r'<script', s)))
    features['pct_non_alnum'] = sum(1 for c in s if not c.isalnum()) / max(1,len(s))
    # add entropy:
    probs = np.array(list(pd.Series(list(s)).value_counts(normalize=True)))
    features['entropy'] = -(probs * np.log2(probs+1e-9)).sum()
    return features

# Build dataset (example)
rows = [
  ({'uri':'/search?q=test', 'body':'', 'headers':{}, 'method':'GET'}, 0),
  ({'uri':'/login', 'body':"user=admin' OR '1'='1", 'headers':{}, 'method':'POST'}, 1),
]
df = pd.DataFrame([ {**extract_features(req), 'label':label} for req,label in rows ])
X = df.drop('label',axis=1)
y = df['label']
Xtr,Xte,ytr,yte = train_test_split(X,y,test_size=0.4,random_state=42)
clf = XGBClassifier(use_label_encoder=False, eval_metric='logloss', n_estimators=20)
clf.fit(Xtr,ytr)
print(classification_report(yte, clf.predict(Xte)))

# Get SHAP explainability
explainer = shap.TreeExplainer(clf)
shap_vals = explainer.shap_values(Xte)
print("SHAP sum example:", shap_vals[0].tolist())
```

**13) Metrics & reporting you must include (final deliverables)**

Per PDF (page 3), include:

- Detection accuracy on labelled test sets and simulated zero-day sets.

Hackathon Challenge 3_WAF

- False positive statistics (per 1000 requests or per day) across baseline traffic.

Hackathon Challenge 3_WAF

- Performance: throughput (req/s), P50/P95/P99 latency for inline model.

Hackathon Challenge 3_WAF

- Explainability examples: show SHAP plots or token highlights for at least 10 alerts.

Hackathon Challenge 3_WAF

- Rule recommendation outputs: list of suggested rules with confidence and examples.

Hackathon Challenge 3_WAF

**14) What to show in your 5-minute demo video**

1. One minute: Problem statement + architecture slide. (show PDF mapping)

Hackathon Challenge 3_WAF

2. Two minutes: Live demo of traffic going through TLS terminated reverse proxy → inline detection flagged in dashboard (show SHAP highlight for the flagged request).
3. One minute: Rule recommendation created from the event and admin approving it (show the ModSecurity rule text).
4. One minute: Performance & metrics dashboard (throughput, FP rate) + brief note on retraining and future improvements. (matches deliverables).

Hackathon Challenge 3_WAF

**15) Risks, adversarial considerations & mitigations**

- **Evasion by obfuscation**: use char-n-gram tokenization and sequence models; include mutation-augmented training; combine multiple feature views.
- **False positives**: adjust threshold, use human approval for blocking rules, provide challenge (CAPTCHA) instead of block.
- **Privacy / encryption limits**: if TLS cannot be terminated, rely on behavioral features and document lower detection capability.
- **Performance bottlenecks**: keep inline model small; precompile models; scale horizontally.

**16) Deliverable checklist (exactly what the PDF asks)**

- Fully Functional ML Module + dashboard + API integration (6.1).

Hackathon Challenge 3_WAF

- Source Code + README (6.2).

Hackathon Challenge 3_WAF

- 5-minute demo video showing detection, rule recs, dashboard (6.3).

Hackathon Challenge 3_WAF

- Technical doc (2–3 pages) covering architecture, data pipeline, ML models (6.4).

Hackathon Challenge 3_WAF

- Logs, metrics, reports with anomalies and FP/FN stats (6.5).

Hackathon Challenge 3_WAF

- Presentation (8–10 slides) summarizing the project (6.6).

Hackathon Challenge 3_WAF

---

If you want, I can take the next concrete step now:

- generate the exact ModSecurity rule templates and a short script that automatically converts a flagged ML token list into a rule; **or**
- produce the 2–3 page technical documentation draft (architecture + ML choices + evaluation plan) ready for submission; **or**
- start a ready-to-run repo skeleton (Docker Compose + NGiNX termination + FastAPI ML service + ModSecurity integration + simple dashboard).

Pick one and I'll produce it immediately (I'll not ask you to wait — I'll generate it now).

Sources

**You said:**

* **Lightweight Real-time Model (inline)** * A fast model (e.g., XGBoost or small neural net exported to ONNX/Treelite) that returns a risk score. Low latency, used for monitor/challenge/block decisions. * **Async Deep Analysis Service** * More expensive models: Transformer / CNN on tokenized bodies, autoencoders for anomalies, graph-based behavioral models (sessions/IP graphs). Run asynchronously; results feed back to learning & rule suggestion engine.

means you are saying implement real-time model as well as async deep analysis service both ?

**ChatGPT said:**

Great question — **No, you are *not required* to implement both in full complexity.**
But here's the correct interpretation for the hackathon and what you *should* build depending on your time, skills, and scoring priorities.

---

### ✅ Short Answer

**Yes — the architecture suggests two components (real-time model + async deep analysis)**
...but
👉 **you can implement a simplified version that still meets all requirements and scores high.**

You **do NOT need a huge deep model**. What's required is the *capability* to analyze suspicious events deeper and generate rule recommendations—not necessarily a big neural net.

---

### ✔️ Why the architecture recommends TWO models

The challenge PDF emphasizes:

**1. Real-time detection must be low-latency**

You cannot put a slow Transformer model inline. Otherwise the WAF stalls.

**2. But they want:**

- Support for supervised, unsupervised, semi-supervised learning
- Explainability
- Continuous learning
- Rule recommendation based on ML insights
- Ability to detect zero-day attacks
- Encrypted traffic handling

These require **heavier processing**, which is better done *asynchronously*, not inline.

Thus, the dual-layer approach is practical and aligns with the PDF's scoring criteria.

---

### ⚡ Practical Hackathon Interpretation

Here is the **correct minimum implementation** that satisfies the requirements:

---

### ✅ 1) Real-Time Lightweight Model (Inline)

**Purpose:**

- Output a **risk score** fast (1–5ms)
- Used for **blocking / monitoring** decisions

**Implementation (easy):**

- XGBoost or LightGBM with engineered features
- Compiled to ONNX/Treelite for speed
- Integrated with WAF (ModSecurity/NGINX/Envoy)

**Why necessary?**

- Directly satisfies:
  ✔ "low latency"
  ✔ "real-time anomaly detection"
  ✔ "baseline behavioral deviation detection"

---

### ✅ 2) Async Deep Analysis Service (Simplified!)

**Purpose:**

Not inline. Runs *after* a suspicious request occurs.

**What it should do (minimal version):**

- Re-analyze flagged request using more detailed features
- Provide **explainability output**
- Generate **rule recommendations**
- Store enriched data for **continuous learning**

**Acceptable implementation for hackathon:**

You do **NOT** need to train a Transformer or GNN.

A valid minimal async analyzer could simply:

- Run SHAP explainability on the XGBoost model

- Perform anomaly score using Isolation Forest or Autoencoder
- Run a deeper regex/token analysis
- Suggest rule templates (ModSecurity rules)
- Update datasets for retraining

**This satisfies the PDF's expectations:**

✔ "Support for supervised, unsupervised, semi-supervised"
✔ "Explainability for each detected traffic pattern"
✔ "Human-readable rule recommendations"
✔ "Administrator feedback loop"
✔ "Continuous learning capability"

---

### 🟢 So, Do You Need Two Models?

**Option A — Best (full score)**

- Inline **XGBoost** model
- Async **Isolation Forest** or **Autoencoder**
- SHAP + rule generator

**Option B — Moderate (still passes)**

- Inline XGBoost
- Async processor *without* ML (e.g., regex + SHAP + rule templates)
- This still satisfies "deep analysis" logically.

**Option C — Minimum viable (bare pass)**

- Only a realtime XGBoost + dashboard + rule generator
- Explain in documentation why async deep model is omitted
- Still "meets pass/fail" because it integrates ML & WAF & dashboard

But **Option A** gives you highest scoring leverage.

---

### 🧠 Why dual-layer improves scoring massively

The scoring rubric includes:

- Detection accuracy

- False positives
- Explainability
- Robustness against novel attacks
- Performance
- Rule recommendation quality

A single inline model cannot perform all of these well.
A two-layer system **scores high in ALL categories**.

---

### 📌 Recommended Implementation (Do This — Simple & Effective)

**Inline (Fast)**

- XGBoost with ~30–50 features
- Served with FastAPI or embedded in ModSecurity

**Async (Simple but effective)**

- Isolation Forest OR Autoencoder
- SHAP explainability
- Attack token extraction
- Rule generator:
    - Converts tokens → ModSecurity rule template
- Store labeled events in a database for retraining

---

### 🚀 Final Verdict

**✔ Yes, the architecture *includes* two modules**

**❗ But you do NOT need two complex ML models**

**✔ You *should* implement:**

- A fast, real-time model
- A lightweight async analysis pipeline with either:
    - Isolation Forest
    - Autoencoder

    - Or just SHAP + rule engine (valid!)

This is enough to win high scores and satisfy every requirement in the PDF.

---

**If you'd like, I can now create:**

- A detailed architecture diagram
- Full code skeleton (inline + async services)
- Rule generator code
- Complete 8–10 slide presentation
- 2–3 page technical documentation draft
- Docker-compose with WAF + ML module + dashboard

Which one should I generate next?

**You said:**

dual-layer approach

how to achieve this>

**ChatGPT said:**

**Dual-layer WAF ML — practical, step-by-step implementation plan**

Great — here's a concrete, runnable plan to **implement the dual-layer approach** (fast inline model + async deep analyzer + rule recommender). I'll give you an ordered checklist, minimal working code snippets, config examples (ModSecurity + FastAPI), testing ideas, deployment notes (Docker Compose), and monitoring + retraining steps. You can follow this end-to-end and have a working demo within a few days.

---

**1) Tech choices (recommended)**

- WAF / proxy: **NGINX + ModSecurity (OWASP CRS)** (easy to demo)
- Inline ML service: **FastAPI** serving a precompiled XGBoost model (ONNX / XGBoost native)
- Async analysis: background worker using **Celery** or a simple queue (Redis) running IsolationForest + SHAP + rule generator
- Storage: **Postgres** (events & labels) + **S3/MinIO** (raw payload archive)
- Message bus: **Redis** (simple), or Kafka if you want scale
- Dashboard: **React** or **Dash/Streamlit** (quick demo)

- Orchestration: **Docker Compose** for hackathon/demo

---

**2) High-level flow (what happens for each request)**

1. Client → TLS terminated at NGINX (so payload is decrypted).
2. NGINX/ModSecurity logs request and calls FastAPI inline ML endpoint (or sends request to sidecar).
3. Inline model (XGBoost) returns score quickly.
   - If score < monitor_threshold → allow.
   - If monitor_threshold ≤ score < block_threshold → mark monitor/challenge (log + let through).
   - If score ≥ block_threshold → either block (in demo, detection mode + suggested block) or return block decision.
4. Suspicious requests (score ≥ monitor_threshold) are pushed to async queue.
5. Async worker re-analyzes using deeper features, runs IsolationForest/autoencoder, computes SHAP, generates suggested ModSecurity rule text, stores suggestions in DB for admin approval.
6. Admin reviews rules on Dashboard → approves → export to ModSecurity rule file / hot reload.

---

**3) Step-by-step implementation**

**Step A — Prep repo & environment**

Create a repo and a docker-compose.yml that brings up: nginx, modsec, ml_service (FastAPI), worker (Celery/worker), redis, postgres, minio, dashboard. Start simple: nginx + ml_service + redis + postgres.

**Step B — Data & synthetic dataset**

- Collect: normal access logs (or generate with wrk/curl), and positive samples from OWASP payloads (SQLi, XSS).
- Create a CSV: columns uri,body,headers_json,method,label. Label positives as 1, others 0.

**Step C — Feature extractor & baseline model**

Write Python to extract fast features (char counts, keyword flags, entropy, length, params count). Train XGBoost.

train_baseline.py (minimal):

```
# pip install pandas xgboost scikit-learn joblib

import pandas as pd, re, numpy as np

from xgboost import XGBClassifier

from sklearn.model_selection import train_test_split

import joblib


def extract_features_row(row):

    s = (str(row['uri']) + ' ' + str(row.get('body',''))).lower()

    features = {

      'uri_len': len(row['uri'] or ''),

      'body_len': len(row.get('body','') or ''),

      'num_params': s.count('='),

      'has_select': int(bool(re.search(r'\bselect\b', s))),

      'has_union': int(bool(re.search(r'\bunion\b', s))),

      'has_script': int(bool(re.search(r'<script', s))),

      'pct_non_alnum': sum(1 for c in s if not c.isalnum()) / max(1, len(s))

    }

    probs = np.array(list(pd.Series(list(s)).value_counts(normalize=True)))

    features['entropy'] = -(probs * np.log2(probs+1e-9)).sum()

    return features


df = pd.read_csv('dataset.csv') # your prepared dataset

X = pd.DataFrame([extract_features_row(r) for _, r in df.iterrows()])

y = df['label']

Xtr,Xte,ytr,yte = train_test_split(X,y,test_size=0.2,random_state=42,stratify=y)
```

```
clf = XGBClassifier(use_label_encoder=False, eval_metric='logloss', n_estimators=100)
clf.fit(Xtr, ytr)
joblib.dump(clf, 'xgb_baseline.joblib')
print("Saved model.")
```

**Step D — FastAPI inline model service**

ml_service/main.py:

```
# pip install fastapi uvicorn pydantic joblib scikit-learn
from fastapi import FastAPI, Request
import joblib, pandas as pd
import uvicorn, re, numpy as np

clf = joblib.load('xgb_baseline.joblib')
app = FastAPI()

def extract_features(req):
  s = (req.get('uri','') + ' ' + req.get('body','')).lower()
  return {
   'uri_len': len(req.get('uri','') or ''),
   'body_len': len(req.get('body','') or ''),
   'num_params': s.count('='),
   'has_select': int(bool(re.search(r'\bselect\b', s))),
   'has_union': int(bool(re.search(r'\bunion\b', s))),
   'has_script': int(bool(re.search(r'<script', s))),
   'pct_non_alnum': sum(1 for c in s if not c.isalnum()) / max(1,len(s)),
   'entropy': 0.0 # compute same as training if needed
  }

@app.post('/score')
```

```
async def score(req_body: dict):
    feat = extract_features(req_body)
    X = pd.DataFrame([feat])
    score = float(clf.predict_proba(X)[0,1])
    # push to redis queue if score >= monitor threshold (example)
    if score >= 0.5:
        # push suspicious event into Redis queue for async analysis
        pass
    return {'score': score, 'features': feat}


if __name__ == "__main__":
    uvicorn.run(app, host='0.0.0.0', port=8000)
```

**Step E — ModSecurity + NGINX integration (call ML service)**

- Configure ModSecurity to **call an external script** or log then let NGINX proxy to ML service. Simpler approach: use NGINX access_log piped to a sidecar that calls ML service and applies local decisions. For hackathon demo, run ModSecurity in **DetectionOnly** and show suggested blocks in dashboard.

Example simplified flow:

- NGINX reverse proxy + access_log to a small Python listener that extracts request JSON and calls /score. If score ≥ threshold, it writes an event with action block_suggested to DB.

You can also use **ModSecurity exec action** to run a lua script that calls the ML endpoint—this is advanced but possible.

**Step F — Async worker (IsolationForest + SHAP + rule generator)**

worker/analyze.py:

```
# pip install scikit-learn shap joblib sqlalchemy redis
from sklearn.ensemble import IsolationForest
import joblib, shap, json, re


iso = IsolationForest(n_estimators=100)
```

```
# train iso on benign features snapshot
# For demo, you can fit on Xtr


def deep_analyze(event):
    # event contains original request, features, model score
    # compute iso anomaly score
    feats = event['features']
    iso_score = iso.decision_function([list(feats.values())])[0]
    # get shap explanation
    # explainer = shap.TreeExplainer(clf)
    # shap_vals = explainer.shap_values(pd.DataFrame([feats]))
    # create rule templates
    suggestions = []
    if event['features']['has_select']:
        suggestions.append({
          'type':'modsec',
          'rule': f"SecRule ARGS|REQUEST_URI \"@contains SELECT\" \"id:1001,phase:2,deny,log,msg:'ML suggested SQLi'\"",
          'confidence': event['score']
        })
    # write back to DB: suggestions, iso_score, shap (optional)
    return {'iso_score': iso_score, 'suggestions': suggestions}
```

**Step G — Rule approval and deployment**

- Dashboard shows suggested rule(s) with example request and SHAP explanation. Admin clicks **Approve** → Backend writes rule into modsec_extra_rules.conf and signals NGINX/ModSecurity to reload (or include file hot-reload). For demo, make reload manual or scripted.

Example rule template (simple):

```
SecRule REQUEST_URI|ARGS "@rx (?i:union\s+select)" \
```

"id:100001,phase:2,deny,log,rev:'1',msg:'ML suggested SQLi pattern',severity:2"

**Step H — Dashboard (alerts + explainability)**

- Show: live alerts, top features (SHAP or list), suggested rules, approve/reject button, and graphs (FP rate, latency).
- Quick option: Streamlit app that queries Postgres for alerts and displays.

---

**4) Docker Compose (toy)**

docker-compose.yml snippet:

```
version: '3.7'

services:

 nginx:

  image: nginx:latest

  ports: ["443:443","80:80"]

  volumes: ["./nginx/conf:/etc/nginx/conf.d","./modsec:/etc/modsecurity"]

 ml_service:

  build: ./ml_service

  ports: ["8000:8000"]

  depends_on: []

 worker:

  build: ./worker

  depends_on: ["ml_service","redis"]

 redis:

  image: redis:alpine

 postgres:

  image: postgres:15

  environment: POSTGRES_PASSWORD: password
```

Start with docker compose up --build.

---

**5) Thresholding & actions (practical)**

- Choose thresholds by validation:
    - monitor_threshold = 0.4 (log + investigate)
    - block_threshold = 0.85 (candidate block; require admin approval for automatic block in demo)
- For demo, prefer **detection mode** (no live blocking) and show suggested blocks; explain in doc how to enable automatic block in production.

---

**6) Testing & evaluation (must include)**

Create scripts to simulate scenarios:

1. **Baseline traffic**: use wrk / hey to generate GET/POST normal requests; record FP rate.
2. **Known attacks**: inject OWASP payloads; measure detection recall.
3. **Obfuscated attacks**: mutate payloads (URL encode, double encode, insert comments) to test robustness.
4. **API abuse / bot**: simulate high request rate from rotating IPs; test session/behavioral detection.

Collect metrics:

- True Positives, False Positives, Precision, Recall, F1.
- Latency: measure P50/P95/P99 for ML service (use wrk or ab on the ml_service endpoint).
- Throughput: requests/sec handled by inline path.

---

**7) Monitoring & retraining**

- Monitor: Prometheus (expose metrics from FastAPI via prometheus_client), Grafana dashboard for latency, score distribution, alerts per minute.
- Retraining pipeline:
    1. Periodically (weekly) export labeled events (approved rules → positive labels; analyst feedback).
    2. Recompute features, retrain XGBoost, validate on time-split test set.

3. If improved (higher recall at acceptable FPR), promote model and deploy (rolling update).

- Use data versioning (commit model artifacts with git + store dataset hash).

---

**8) Demo & deliverables checklist (what to show)**

- Live demo: send a malicious request → show ML service score → show async suggestion → approve rule → show rule text.
- Include: source code, README.md with start instructions, docker-compose.yml, dataset.csv (synthetic), technical_doc.pdf (2–3 pages), demo video (recorded), slides (8–10), and logs/metrics CSV.

---

**9) Quick pitfalls & mitigations**

- **High false positives**: tune thresholds; use monitoring to show FP per 1k requests and request human approval before blocking.
- **TLS termination unavailable**: document and use metadata-only detection (rate, JA3-like fingerprinting) — note reduced capability.
- **Latency too high**: precompile model, use smaller models and fewer features inline, move heavy work to async.
- **Data privacy**: sanitize or redact sensitive fields before storage.

---

**10) Next step code deliverable I can produce now**

Pick one (I will generate immediately):

1. Full docker-compose.yml + minimal working FastAPI ml_service + sample NGINX config that logs and forwards requests to ML service.
2. Complete train_baseline.py + a synthetic dataset.csv with 200 rows (mix of benign + attack) so you can train and test locally.
3. worker/analyze.py + rule generator function that converts top tokens to ModSecurity rules.
4. 2–3 page technical doc draft (ready to submit).
5. 8–10 slide deck outline + speaker notes for the 5-min demo.

Which of the five should I produce now? (I'll generate it in this reply.)