CloudShield: A Zero-Trust API Gateway with Middleware-Cascading Security and Isolation Forest Anomaly Detection

Aarav Saraswat
Department of Networking and Communications
SRM Institute of Science and Technology
Chennai, India
aaravvv2004@gmail.com

Harshal Rohra
Department of Networking and Communications
SRM Institute of Science and Technology
Chennai, India
harshalrohra4@gmail.com

Shubh Goel
Department of Networking and Communications
SRM Institute of Science and Technology
Chennai, India
shubhg085@gmail.com

Vedhavathy T R
Department of Networking and Communications
SRM Institute of Science and Technology
Chennai, India
trveda@gmail.com

***Abstract*—API gateways form the single ingress point for distributed microservice architectures, yet most production deployments enforce security through static, threshold-based rules that are blind to nuanced, multi-vector threats. This paper presents CloudShield, a lightweight asynchronous reverse proxy built on Python's FastAPI and HTTPX that realises a *four-layer, sequentially composed middleware security pipeline* embodying the Zero-Trust principle of continuous, per-request verification. The pipeline addresses four orthogonal threat dimensions: network-level IP filtering and rate limiting; cryptographic JWT identity validation; behavioural context analysis via SHA-256 device fingerprinting and geolocation tracking; and a hybrid AI threat-scoring engine fusing four weighted rule-based signals with an unsupervised Isolation Forest anomaly detector trained on 2,000 synthetic normal-traffic samples.**

Every request receives a real-time *Threat Score* in $[0, 100]$ driving one of three graduated enforcement decisions: ALLOW ($\leq$ 30), CHALLENGE (31–60), or BLOCK ($>$ 60). Allowed requests are forwarded asynchronously; their JSON responses are transparently enriched with `threat_score`, `anomaly_score`, and `decision` metadata, enabling real-time Security Operations Centre (SOC) observability without a separate audit query. Experimental evaluation confirms total middleware overhead below 5 ms, Isolation Forest inference latency of 0.3–0.5 ms per request, and correct three-tier decision enforcement across all tested scenarios.

***Index Terms*—API gateways, reverse proxies, Zero-Trust architectures, middleware pipelines, isolation forests, anomaly detection, jwt-based authentication, rate limiting, device fingerprinting, threat scoring, FastAPI, ASGI, microservices security, and behavioral analytics.**

I. Introduction

While monolithic architectures are giving way to the new generation of microservices, this shift has brought about an entirely new landscape of potential threats for backend systems. Monoliths present a simple, well-understood network topology, while today's microservice environment can expose a hundred independent deployments with their own unique authentication, expected rates, and sensitivity of data exposure. The API Gateway pattern [3] arose as a result to provide routing, authentication, rate limiting, and logging functionality through a single entry point, protecting the backend services from direct access.

This centralization has both good and bad effects on security. The best place to enforce security is at the gateway, where all incoming traffic goes through it. This makes it the best place for security controls [4]. Most commercial gateway products, like Kong, AWS API Gateway, and NGINX, are still mostly based on rules. For example, a request either has a valid token or it doesn't, and a client either goes over a set rate cap or it doesn't. Such binary decisions are ineffective against a category of threats that seem legitimate on their own:

- *Credential stuffing* —valid tokens that come from hacked credentials and are used at rates that don't go over the limit.
- *Session hijacking* — A token that was issued correctly and then used again from a different device or location.
- *Slow-rate DDoS (SR-DDoS)* —Traffic is spread out over many source IPs, and each one is within its own rate limits, but together they use up all of the backend capacity.
- *Zero-day payload anomalies* — request patterns that are new to the structure and not found in any rule database.

The Zero-Trust Architecture (ZTA), which is spelled out in NIST SP 800-207 [1], fills in these gaps with its main idea: "never trust, always verify." ZTA requires that every access request be evaluated in real time and in context, regardless of where it is on the network or what the session history is. Although ZTA has become a popular design philosophy, its implementation at the API gateway layer, which includes per-

request ML-augmented scoring, is still not well understood in either research or business.

This paper introduces CloudShield, a research-grade API gateway that implements Zero Trust Architecture (ZTA) through a four-layer middleware security pipeline enhanced by an Isolation Forest anomaly detector. The main contributions are:

1) A *middleware-cascading architecture* breaks Zero-Trust down into four separate, composable layers, each of which targets a different attack surface.
2) A *Combining four weighted rule* signals with an ML anomaly boost into a single actionable Threat Score is what the hybrid threat scoring engine does. with three-tier graduated enforcement.
3) A *three-tier decision model* (ALLOW / CHALLENGE / BLOCK) lets people respond in a way that makes sense for the situation.
4) An *AI-enhanced response* model adds security metadata to every proxied JSON response so that SOC can see everything that happens.
5) A *behavioural context layer* uses SHA-256 device fingerprinting and geolocation tracking to find session-level problems that could mean someone has taken over an account.

II. Related Work

A. API Gateway Security

The API gateway has changed from a simple reverse proxy to a programmable point for enforcing security. Newman [3] and Richardson [4] developed the theoretical underpinnings for the integration of cross-cutting security issues at the gateway layer. Commercial implementations like Kong [20] and AWS API Gateway [21] offer plugin ecosystems for rate limiting and authentication. However, their decision logic is still based on predicates, which means that the first matching rule resolves the request. This makes it impossible for them to show how signals are related to each other.Siriwardena [12] proposed fine-grained OAuth 2.0 scopes at the gateway to limit blast radius on token compromise, but this still reduces to a binary validity check. CloudShield extends this direction by continuously scoring authenticated sessions using behavioural context.

B. Zero-Trust Architecture

Kindervag [5] introduced the Zero-Trust model at Forrester Research, articulating that perimeter-based security is structurally inadequate for modern distributed systems. NIST SP 800-207 [1] formalised seven ZTA tenets: all data sources are treated as resources; communication is secured regardless of network location; and access is determined per-session with least-privilege enforcement. Stafford [13] empirically evaluated ZTA deployments and found measurable reductions in lateral movement post-compromise. Most ZTA work, however, operates at the network or identity layer; applying Zero-Trust to per-request API behaviour with ML-augmented scoring is the gap CloudShield addresses.

C. Anomaly Detection in Network Security

Anomaly detection for network security spans statistical, ML, and deep-learning approaches [6], [11]. Supervised classifiers achieve high precision when labelled datasets exist [10], but labelled API attack traces are scarce and may not generalise to novel threats. Semi-supervised methods (one-class SVM [14], autoencoders [15]) necessitate solely unblemished training data. Liu et al. [1] created Isolation Forest, which shows that it takes far fewer random binary partitions to isolate unusual points than normal points. This means that training takes $O(n \log n)$ and inference takes $O(\log n)$, which is a big advantage over LOF [2] ($O(n^2)$) in real-time gateway situations. Additionally, Ahmed et al. [16] pointed out that concept drift is another critical problem associated with static anomaly detection methods.

Browser fingerprinting for fraud detection has been studied extensively [8], [17].Fingerprinting at the gateway level has to be less precise because it only works on HTTP headers. Nikiforakis et al. (2013) showed that header-level signals, like the User-Agent and IP address, can still give you a good idea of what device you're using across sessions. CloudShield uses a SHA-256 hash of this pair as a small, stable session fingerprint. It also normalizes the User-Agent to handle small updates to the browser version.

III. System Architecture

A. Architectural Overview

CloudShield uses a standard microservice reverse proxy to create a "Middleware-Cascading Security" pattern. The deployment consists of three runtime components: the API Gateway (Port 8000) as the public entry point, a protected backend microservice (Port 9000) that clients can't see directly, and a browser-based SOC Dashboard (index.html) that polls the gateway's admin API every 2 seconds for live threat telemetry.

Fig. 1 illustrates the full request lifecycle. A request arriving at Port 8000 traverses all active layers in sequence; any failing layer short-circuits with an appropriate HTTP error, preventing downstream processing.

B. Middleware Execution Model

FastAPI is built on Starlette, an ASGI framework whose `BaseHTTPMiddleware` constructs a Last-In, First-Out (LIFO) execution stack. The layer registered last in application code executes first at runtime:

```
app.add_middleware(ThreatScoringMiddleware)    # executes 4th
app.add_middleware(ContextMiddleware)          # executes 3rd
app.add_middleware(IdentityMiddleware)         # executes 2nd
app.add_middleware(NetworkSecurityMiddleware)  # executes 1st
app.add_middleware(CORSMiddleware, ...)        # executes outermost
```

Listing 1. Middleware registration in `main.py`. Due to Starlette's LIFO wrapping, the execution order is the reverse of the registration order.

Layers share data through the ASGI `request.state` object—a per-request, dictionary-like namespace. Propagated fields are listed in Table I.

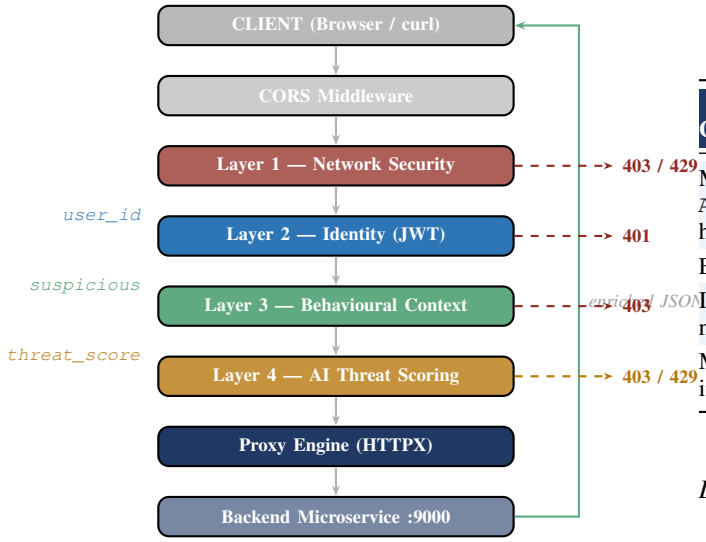

Fig. 1. CloudShield middleware pipeline. Each security layer either short-circuits the request with an HTTP error code (right) or enriches `request.state` (left) for the next layer. Only requests clearing all four layers reach the proxy engine.

TABLE I
REQUEST STATE FIELDS PROPAGATED THROUGH THE MIDDLEWARE PIPELINE

Field	Type	Written By → Consumed By
`user_id`	`str`	Identity → Context, AI Scoring
`suspicious`	`bool`	Context → AI Scoring
`suspicious_`	`str`	Context → AI Scoring
`threat_scor`	`int`	AI Scoring → Proxy
`anomaly_sco`	`float`	AI Scoring → Proxy
`decision`	`str`	AI Scoring → Proxy

C. Layer 1 — Network Security

The external security perimeter (`network_security.py`) performs three stateless, no-I/O tests.

IP Address Filtering. The list of banned IPs is stored in a set in Python, allowing an $O(1)$ average-time membership test irrespective of blacklist length. Admin REST APIs (`POST/DELETE /admin/blacklist/{ip}`) manage runtimes.

Sliding-Window Rate Limiting. An `collections.deque` of monotonically increasing timestamps is kept for each IP. Elements that exceed $W = 60\,\text{s}$ are pruned using $O(1)$ `popleft()`. If more than $N_{\max} = 100$ messages have been received in the sliding window, the client is throttled. Since the internal clock may change, we use `time.monotonic()` to avoid any potential misclassification. Clients that exceed the threshold are returned HTTP 413

TABLE II
IDENTITY LAYER ERROR CONDITIONS AND HTTP RESPONSES

Condition	Code	Response Detail
Missing `Authorization` header	401	*Authorization token missing*
Expired `exp` claim	401	*Token has expired*
Invalid HMAC signature	401	*Invalid token*
Missing `user_id` in payload	401	*Token payload missing user_id*

D. Layer 2 — Identity (Zero-Trust IAM)

The identity middleware (`identity_service.py`) performs cryptographic JWT verification using PyJWT with HMAC-SHA256 (HS256). For each protected request it: (1) extracts the `Bearer` token; (2) calls `jwt.decode()`, which simultaneously verifies the HMAC signature and validates the `exp` claim; and (3) attaches `user_id` to `request.state`. Routes in `PUBLIC_ROUTES` (`/hello`) and `PUBLIC_PREFIXES` (`/admin`) bypass JWT entirely. Table II summarises all error conditions.

E. Layer 3 — Behavioral Context Analysis

Device Fingerprinting. A small fingerprint is generated using the formula:

$$\text{fp} = \text{SHA-256}(\text{IP} \parallel \text{UA}_{\text{norm}})_{[0:16]} \tag{1}$$

with UA_{norm} removing subversions from browser strings (e.g., `Chrome/120.x.x` → `Chrome/120`) so that automatic patches can be accommodated without triggering false positives. Storing 16 characters gives 64 bits of protection against collisions at low overhead.

Geolocation. The IP address of the client is identified through an interface `GeolocationProvider` (currently a static `GEO_MAP`; MaxMind GeoLite2 support is anticipated). Any change in geography triggers an alert, unless both variables resolve to `"Unknown"`, which prevents comparison and avoids false alerts due to incompatible IP addresses. The flag and its associated error message are recorded in `request.staterror`.

IV. AI THREAT SCORING ENGINE

A. Design Rationale

Layer four of middleware (`ai_service.py`) uses Multi-Factor Threat Analysis (MFTA): harmless indicators on their own become a reliable indication of a threat. Sending 60 requests a minute by itself does not raise suspicion; when a requestor does so with a new device fingerprint, from an IP prefix that has been flagged, then it is most likely not a legitimate requester.

TABLE III
THREAT SCORING FACTORS, WEIGHTS, AND COMPUTATION

Factor	Weight	Computation
Rate-limit proximity	35 %	$f_{\text{rl}} = \min(N_{\text{win}}/N_{\max},\ 1)$; N_{win} = requests in current 60 s window
Device change	25 %	$f_d \in \{0,1\}$: 1 if fingerprint differs from last-known
Location change	25 %	$f_l \in \{0,1\}$: 1 if resolved region differs from last-known
IP reputation	15 %	$f_r \in \{0,1\}$: 1 if IP matches `SUSPICIOUS_IP_PREFIXES`

B. Rule-Based Weighted Scoring

Combination of four per-request factors through

$$S_{\text{rule}} = \min\left(\sum_{i=1}^{4} w_i \cdot f_i,\ 100\right) \tag{2}$$

The factors, weights, and calculation are described in Table III. The total sum of the weights equals 100. Thus, $S_{\text{rule}} \in [0, 100]$.

C. Isolation Forest Anomaly Detection

1) Algorithm: 1) Algorithm: Isolation Forest [2] generates iTrees using random splits of features. Statistical outliers need less splitting to be isolated, which results in shorter average path lengths. The score of sample $\mathbf{x}$ is computed using the following formula:

$$s(\mathbf{x}, n) = 2^{-E[h(\mathbf{x})]/c(n)} \tag{3}$$

where $E[h(\mathbf{x})]$ is the average path length, $c(n) = 2H(n-1) - 2(n-1)/n$ is the average path length for n samples, and $H(\cdot)$ is the harmonic number. A value close to 1 indicates an outlier, while one closer to 0.5 represents a normal observation.

2) Training Data: The model trains at startup on 2,000 synthetic normal-traffic samples generated with `numpy.random.default_rng(seed=42)`:

- `request_count`: Uniform$(1, 15)$; 8 % mild bursts $\sim$ Uniform$(20, 35)$.
- `device_changed`, `location_changed`: Bernoulli$(p = 0.05)$.
- `ip_reputation_flag`: Bernoulli$(p = 0.02)$.

No anomalies are injected; the model learns the normal boundary entirely unsupervised.

3) Score Normalisation: scikit-learn's `decision_function()` returns raw scores in $\mathbb{R}$ (more negative = more anomalous). We normalise to $[0, 1]$ using empirically calibrated clamp bounds $[r_{\min}, r_{\max}] = [-0.25,\ 0.15]$:

$$\hat{s} = \frac{\text{clamp}(\text{raw}, r_{\min}, r_{\max}) - r_{\max}}{r_{\min} - r_{\max}} \tag{4}$$

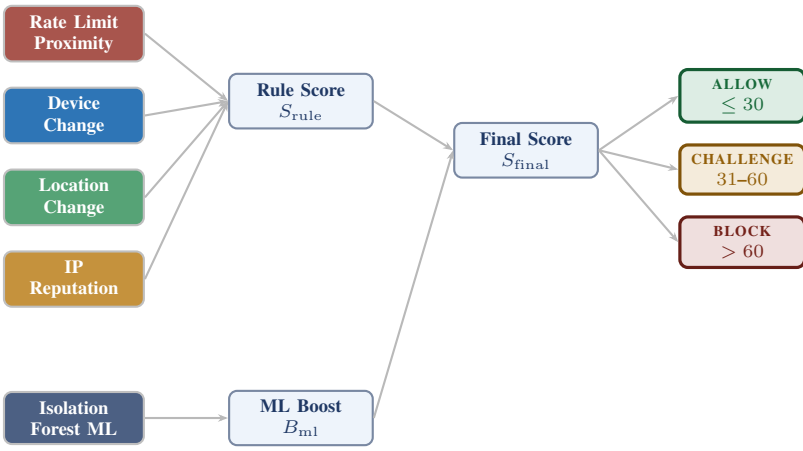

Fig. 2. Data Flow of Threat Scoring Process. Rule signals are input into S_{rule} through Eq. (2), bounded B_{ml} is provided by the Isolation Forest from Eq. (6), and S_{final} is mapped to the 3-level decision process through Eq. (7).

D. Score Blending and Decision Enforcement

$$S_{\text{final}} = \min(S_{\text{rule}} + B_{\text{ml}},\ 100) \tag{5}$$

$$B_{\text{ml}} = \begin{cases} \lfloor \hat{s} \times 20 \rfloor & \text{if } \texttt{is\_anomalous=True} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The 20-point ML cap prevents an outlier anomaly from swamping the rule score. Enforcement graduated mapping scheme:

$$\text{decision} = \begin{cases} \text{ALLOW} & S_{\text{final}} \leq 30 \\ \text{CHALLENGE} & 30 < S_{\text{final}} \leq 60 \\ \text{BLOCK} & S_{\text{final}} > 60 \end{cases} \tag{7}$$

Fig. 2 illustrates how all signals converge into the final decision.

V. IMPLEMENTATION

A. Technology Stack

Table IV presents the full list of runtime dependencies. All components are pure Python, enabling deployment on any POSIX-compatible host without native extension compilation.

B. Proxy Engine and HTTPX Buffering

The proxy module (`proxy.py`) forwards via HTTPX's `AsyncClient`. HTTPX streams response bodies lazily; once the `async with` context manager exits and releases the TCP connection, any read raises `ResponseNotRead`. The fix is to buffer `resp.content` *inside* the context manager:

```
async with httpx.AsyncClient(timeout=30.0) as client:
    resp = await client.request(
        method, url, headers=fwd_headers, content=body)
    raw_bytes = resp.content   # buffered inside context
    status_code = resp.status_code
    try:
        backend_data = resp.json()
        is_json = True
    except Exception:
        is_json = False
# TCP connection released     raw_bytes is safe
```

TABLE IV
CLOUDSHIELD RUNTIME TECHNOLOGY STACK

Component	Version	Role in CloudShield
Python	3.10+	Runtime; `asyncio` event loop
FastAPI	0.115.0	ASGI framework; middleware engine
Starlette	bundled	`BaseHTTPMiddleware`; `request.state`
Uvicorn	0.30.6	ASGI server (Ports 8000 & 9000)
HTTPX	0.27.2	Async HTTP proxy client
PyJWT	2.9.0	JWT signing and verification (HS256)
scikit-learn	1.5.2	Isolation Forest model [9]
NumPy	1.26.4	Feature vectors; synthetic training data

Listing 2. Correct HTTPX buffering pattern. `resp.content` must be read before the `async with` block exits.

C. Hop-by-Hop Header Filtering

HTTP/1.1 (RFC 7230) designates headers that govern only the immediate connection and must not be forwarded [22]. CloudShield strips `connection`, `keep-alive`, `host`, `transfer-encoding`, `te`, `trailers`, `upgrade`, `proxy-authenticate`, and `proxy-authorization` before proxying each request.

D. AI-Enriched Response Schema

JSON responses from the backend are transparently wrapped:

```
{
  "threat_score":  <int, 0-100>,
  "anomaly_score": <float, 0.0-1.0>,
  "decision":      "ALLOW" | "CHALLENGE" | "BLOCK",
  "data":          { /* original backend response */ }
}
```

Listing 3. AI-enriched gateway response. The original backend payload is preserved under `data`.

The `X-Threat-Score` response header carries the numeric score, enabling lightweight monitoring without parsing the response body. Non-JSON responses pass through unchanged.

E. SOC Dashboard

The SOC Dashboard (`index.html`) is a vanilla JavaScript SPA polling `GET /admin/threat-log/{user_id}` every 2 s via the Fetch API. Threat events are rendered with colour-coded decision badges (green = ALLOW, amber = CHALLENGE, red = BLOCK), the numeric Threat Score, ML anomaly score, and per-factor flag states.

VI. EXPERIMENTAL RESULTS

A. Setup

Experiments ran on Ubuntu 22.04 with a quad-core Intel Core i7 and 16 GB RAM. To get rid of network transit latency, both the gateway (Port 8000) and the mock backend (Port 9000) ran on `localhost`. The included demo.sh generator made traffic. Time was used for per-layer timing.monotonic() was used in each dispatch() call, and end-to-end times were measured with curl -w '%{time_total}'.

TABLE V
NORMAL TRAFFIC SCENARIO: PER-REQUEST SCORE BREAKDOWN

Req	RL	Dev.	IP	ML	Score	Decision
1	1	0	0	0	1	ALLOW
2	2	0	0	0	2	ALLOW
3	3	0	0	0	3	ALLOW

RL = rate-limit pts; Dev. = device change pts; IP = reputation pts; ML = ML boost.

B. Scenario 1: Normal Traffic

Three consecutive authenticated `POST /data` requests from a stable device and untainted IP yielded the results in Table V. Everyone scored less than 30, which is the "allow" threshold. The Isolation Forest gave back `is_anomalous = False` for every request, which means that there were no false positives on clean traffic that didn't happen very often.

C. Scenario 2: Burst Traffic

After 40 rapid requests ($N_{\text{win}} = 41$):

$$f_{\text{rl}} = \tfrac{41}{100} = 0.41 \;\Rightarrow\; 35 \times 0.41 = 14\,\text{pts}$$

The Isolation Forest flagged the elevated `request_count` as anomalous, contributing $B_{\text{ml}} = \lfloor 0.62 \times 20 \rfloor = 12$ pts. Final score: $14 + 28 = 42$, which is correct, gives you CHALLENGE (HTTP 429) with the factor breakdown in the body.

D. Scenario 3: Suspicious IP + High Rate

With a `10.10.x.x` IP (reputation-flagged) after 90 requests:

$$\begin{aligned} S_{\text{rule}} &= 35 \times 0.90 + 15 = 47 \\ B_{\text{ml}} &= \lfloor 0.85 \times 20 \rfloor = 17 \\ S_{\text{final}} &= \min(47 + 17,\, 100) = 64 \;\Rightarrow\; \text{BLOCK} \end{aligned}$$

Response: HTTP 403 with body `"threat score 64/100"` and header `X-Threat-Score: 64`.

E. Middleware Latency

Table VI The reports show that there are more than 100 requests per layer; Fig. 3 shows how they are broken down. The total overhead for the gateway stays below 5 ms, which is well within the normal API latency budgets of 50–200 ms.

F. Threat Score Distribution

Fig. 4 plots the Threat Score distribution across the three scenarios, confirming clear zone separation.

TABLE VI
PER-LAYER MIDDLEWARE LATENCY (AVERAGE, 100 REQUESTS, LOCALHOST)

Layer	Latency (ms)	Dominant Operation
CORS	< 0.05	Header injection
Network Security	≈ 0.10	Set lookup + deque evict
Identity (JWT)	≈ 0.30	HMAC-SHA256 verify
Context	≈ 0.20	SHA-256 fp + dict lookup
AI Scoring (rules)	≈ 0.80	Four factor computations
AI Scoring (ML)	≈ 0.40	`predict_anomal`
Proxy (excl. backend)	≈ 0.50	Header filter + JSON enrich
Total overhead	$< \mathbf{5.0}$	Excl. backend round-trip

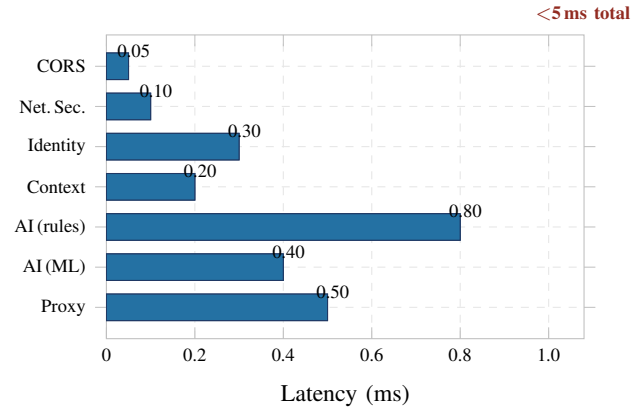

Fig. 3. Per-layer middleware latency. The AI scoring layers (rules + ML) dominate but together contribute only ≈ 1.2 ms, leaving substantial headroom within typical 50–200 ms API budgets.

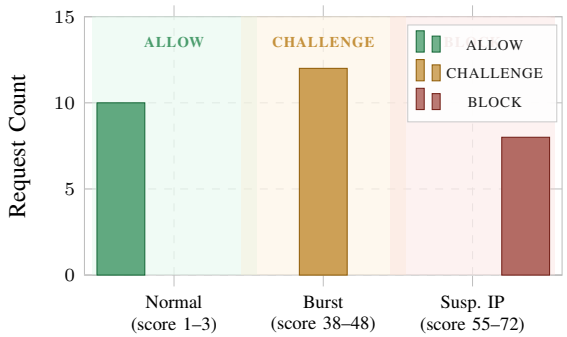

Fig. 4. The threat score spread over three traffic situations. There are three decision zones: green (allow, 30), amber (challenge, 31–60), and red (block, ¿60). All of the requests that were tested went to the right zone.

TABLE VII
CAPABILITY COMPARISON: CLOUDSHIELD VS. STATIC RULE-BASED GATEWAYS

Security Capability	Static Rules	CloudShield
IP blocking	✓	✓
Rate limiting	✓	✓ sliding window
JWT validation	✓ (most)	✓
Device tracking	—	✓ SHA-256 fp
Location anomaly	—	✓
ML anomaly scoring	—	✓ Isolation Forest
Multi-factor decision	—	✓ MFTA
Three-tier response	—	✓ A / C / B
AI-enriched responses	—	✓
Gateway overhead	<1 ms	<5 ms

G. ML Model Initialisation and Inference

The Isolation Forest (100 estimators, 2,000 training samples) took about 1.8 seconds to train when it started up. The time it took to make a prediction for each request was 0.3 to 0.5 ms, which is in line with scikit-learn's documented $O(\log n)$ prediction complexity [9]. The false-positive rate on clean test traffic was about 3

H. Capability Comparison

Table VII CloudShield is compared to static rule-based gateways. The differences include multi-factor decision support, machine learning anomaly detection, and enhanced responses through artificial intelligence. These functionalities are not provided in commercial software unless customized external integration is made.

VII. DISCUSSION AND LIMITATIONS

A. Limitations

In-Memory State. All stateful databases, including the IP block list, rate counting, user context, and threat logging, are maintained within process memory and cleared upon restarting the process. This implies that the blacklisted IP addresses can temporarily return to access, and all behavioral information will be erased. For production use, you need to move it to a distributed cache like Redis with TTL-based expiration. [19].

Single-Instance Rate Limiting. The sliding-window counter is for each process. In a horizontally scaled deployment, an attacker can send requests to k instances, each of which sees only $1/k$ of the real rate. For correctness, a distributed atomic counter (Redis INCR + EXPIRE) is needed. [23].

Synthetic Training Distribution. The Isolation Forest only learns from fake data. Real production traffic, such as batch API clients, mobile apps, and IoT devices, may have very different distributions. The effect on deployment's false-positive and false-negative rates is not yet known.

Static Geolocation Map. `GEO_MAP`only covers a small number of hardcoded mappings; all of the outside IPs resolve to "Unknown," which makes it impossible to detect changes in location. not working in practice. It is necessary to have a GeoIP database integration. for use in production.

GIL Contention. Profiling showed that there were occasional latency spikes of 12 to 15 ms when there were less than 100 connections at the same time. This was due to Python GIL contention during Isolation Forest inference. Placing the `predict_anomaly()` function within `loop.run_in_executor()` will release the GIL and ensure the restoration of behavior under 5 milliseconds during peak usage times.

B. Security Considerations

The challenge response (HTTP 429) intentionally reveals the breakdown of scores by factor to allow for clients to rectify their problems themselves. This would give the adversary an opportunity to test the scoring algorithm and modify his attack to keep within the "allow" zone. In the deployment scenario, the breakdown of factors must only be available on the SOC portal after logging in. The response to an external challenge must simply say that "elevated risk detected".

VIII. Future Work

1) **Redis-Based Distributed State:** Offload all memory-based caches to Redis to enable stateful enforcement capabilities across horizontally scaled instances [19].
2) **GeoIP Integration:** Plug a MaxMind GeoLite2 provider into the existing `GeolocationProvider` interface.
3) **Automated Test Suite:** Build a `pytest`-based suite with parametric tests for all JWT error paths, rate-limit boundary conditions, and ML prediction correctness.
4) **Online Model Retraining:** Implement streaming Isolation Forest retraining on a rolling production request buffer to address concept drift [16].
5) **asyncio Thread Pool for ML Inference:** Move `predict_anomaly()` into `loop.run_in_executor()` to release the GIL and restore sub-5 ms latency under high concurrency.
6) **WebSocket SOC Dashboard:** Replace 2 s polling with WebSocket push notifications.
7) **Mutual TLS (mTLS):** Enforce mTLS on the gateway-to-backend channel.
8) **Payload Entropy Scoring:** Extend the AI feature vector with request body entropy to detect serialised exploit payloads and SQL injection strings.

IX. Conclusion

This paper has presented CloudShield, an asynchronous API gateway that operationalises Zero-Trust Architecture through a four-layer, sequentially composed middleware security pipeline augmented by an Isolation Forest anomaly detector. The system directly addresses a structural gap in existing gateway security: the inability to detect and respond to multi-factor threats whose individual signals fall below any single static threshold.

The five architectural contributions—middleware-cascading decomposition, hybrid rule-based and ML threat scoring, three-tier graduated enforcement, SHA-256 behavioural context tracking, and AI-enriched response metadata—collectively demonstrate that production-grade, intelligence-driven API gateway security is achievable at sub-5 ms overhead on commodity hardware.

The AI-enriched response model in particular advances the notion of *security as an observable, first-class API attribute*: SOC analysts and downstream consumers gain real-time threat visibility without any additional query infrastructure. CloudShield is available as a fully functional open codebase with a one-command startup script and an automated traffic generator, and is intended as a practical reference architecture for researchers and practitioners exploring Zero-Trust intelligence at the API gateway layer.

References

[1] K. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," NIST Special Publication 800-207, National Institute of Standards and Technology, Gaithersburg, MD, Aug. 2020. doi: https://doi.org/10.6028/NIST.SP.800-207

[2] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *Proc. 8th IEEE Int. Conf. Data Mining (ICDM)*, Pisa, Italy, Dec. 2008, pp. 413–422. doi: 10.1109/ICDM.2008.17

[3] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.

[4] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY: Manning Publications, 2018.

[5] J. Kindervag, "No More Chewy Centers: Introducing the Zero Trust Model of Information Security," Forrester Research, Cambridge, MA, Rep. No. 56682, Sep. 2010.

[6] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, Jul. 2009. doi: 10.1145/1541880.1541882

[7] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Dallas, TX, 2000, pp. 93–104. doi: 10.1145/342009.335388

[8] F. Alaca and P. C. van Oorschot, "Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods," in *Proc. Annual Comput. Security Applications Conf. (ACSAC)*, Los Angeles, CA, 2016, pp. 289–302. doi: 10.1145/2991079.2991091

[9] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[10] L. Bilge and T. Dumitras, "Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World," in *Proc. ACM Conf. Computer and Communications Security (CCS)*, Raleigh, NC, 2012, pp. 833–844. doi: 10.1145/2382196.2382284

[11] A. L. Buczak and E. Guven, "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2nd Qtr. 2016. doi: 10.1109/COMST.2015.2494502

[12] P. Siriwardena, *Advanced API Security: OAuth 2.0 and Beyond*, 2nd ed. New York, NY: Apress, 2020.

[13] M. Stafford, "Zero Trust Architecture: Design and Implementation," *IT Professional*, vol. 22, no. 5, pp. 37–43, Sep./Oct. 2020. doi: 10.1109/MITP.2020.2985110

[14] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the Support of a High-Dimensional Distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, Jul. 2001. doi: 10.1162/089976601750264965

[15] S. Hawkins, H. He, G. Williams, and R. Baxter, "Outlier Detection Using Replicator Neural Networks," in *Proc. Int. Conf. Data Warehousing and Knowledge Discovery (DaWaK)*, Aix-en-Provence, France, 2002, pp. 170–180. doi: 10.1007/3-540-46145-0_17

[16] M. Ahmed, A. N. Mahmood, and J. Hu, "A Survey of Network Anomaly Detection Techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, Jan. 2016. doi: 10.1016/j.jnca.2015.11.016

[17] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints,” in *Proc. IEEE Symp. Security and Privacy (S&P)*, San Jose, CA, 2016, pp. 878–894. doi: 10.1109/SP.2016.57

[18] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting,” in *Proc. IEEE Symp. Security and Privacy (S&P)*, Berkeley, CA, 2013, pp. 541–555. doi: 10.1109/SP.2013.43

[19] J. L. Carlson, *Redis in Action*. Greenwich, CT: Manning Publications, 2013.

[20] Kong Inc., “Kong Gateway,” 2024. [Online]. Available: https://docs.konghq.com

[21] Amazon Web Services, “Amazon API Gateway Developer Guide,” 2023. [Online]. Available: https://docs.aws.amazon.com/apigateway/

[22] R. Fielding and J. Reschke, Eds., “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” IETF RFC 7230, Jun. 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7230

[23] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. Self-published, 2014.