IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

AI-Driven Zero Trust Security-As-A-Service:A Gateway-Centric Architecture With Isolation Forest-Based Continuous Trust Evaluation For Cloud-Native Applications

Dr. G. Janaka Sudha[1], Purushothaman R[2]

[1]Associate Professor, [2]UG Student, [1,2]Department of Computer Science and Engineering, [1,2]Sri Venkateswara College of Engineering,

Pennalur, Sriperumbudur, Kancheepuram Dt, Tamil Nadu, India - 602117 [1]

Doi : https://doi.org/10.56975/ijcrt.v14i7.309006

Abstract. Cloud-native back-ends fail open in a particular and uncomfortable way. Once a user has cleared the login form and a JWT has been minted, the request path stops asking questions. Role-Based Access Control (RBAC) treats every subsequent call as equally trustworthy, even when the behaviour around it has shifted in ways an operator would notice instantly. This paper describes ZTaaS, an AI-driven Zero Trust Security-as-a- Service platform built around an external reverse-proxy gateway that re-evaluates trust on every hop. ZTaaS combines a Node.js gateway that performs JWKS-based RS256 verification, runtime policy evaluation and short-lived internal token translation; a continuous telemetry pipeline that captures behavioural features into MongoDB and recomputes per-tenant baselines on a scheduled job; a low-latency deviation-based risk score in the request path paired with an asynchronous Isolation Forest model running off a RabbitMQ queue; and an adaptive enforcement layer that maps the resulting score onto allow, step-up, or block actions without touching the protected back-end. We describe the multi-window feature extraction (60 s, 10 min, 60 min) used to keep freshly authenticated users out of the false-positive bucket, the weighted deviation score, and the unsupervised model bootstrapped on a synthetic baseline of normal traffic with a small number of injected anomalies. The implementation runs in Node.js, Python and React. End-to-end tests show that authentication and authorisation can be enforced as a single source of truth at the edge, that high-risk sessions are short-circuited at the proxy boundary, and that the analytics fan-out adds no measurable latency to the request path under nominal load.

Keywords: Zero Trust Architecture, Security-as-a-Service, API Gateway, Continuous Authentication, Anomaly Detection, Isolation Forest, Cloud-Native Security, Policy-as-Code

1 Introduction

Modern back-ends are mostly seams. A typical cloud-native deployment fans out across dozens of services, each one reachable through a documented HTTP API and several less documented internal ones. Authentication is delegated to an identity provider; authorisation is bolted into the application code in pieces that have grown over time. The result is a system in which trust is established at login and then assumed to persist for the lifetime of a token. That assumption breaks the moment a credential is phished, a refresh token is replayed, or an authenticated session is hijacked from a second device [1, 4].

Zero Trust is, in principle, the answer: never trust, always verify [10]. In practice, the principle has to be operationalised without a rewrite of the protected workload. Existing offerings either ship as SDKs

that the application embeds (intrusive), or as vendor-locked control planes wired to a single cloud, or as narrowly scoped point solutions for one signal such as device posture [3, 7]. The gap that motivated this work is the absence of an external, application-independent gateway that performs the ongoing trust evaluation itself, scores user behaviour with a lightweight ML pipeline, and translates the result into policy-driven enforcement without asking the back-end to change.

ZTaaS sits in front of an unmodified HTTP back-end and takes over every per-request security decision. External JWTs are verified through a cached JWKS; declarative authorisation policies are evaluated per tenant; behavioural features are streamed to MongoDB; and a deviation-based score is computed on the request path. The same feature vector is published asynchronously to a Python service that runs an Isolation Forest [9] and writes a richer anomaly record back to the data store. The score is mapped through tenant-specific thresholds—defaulting to 0.7 for HIGH and 0.4 for MEDIUM— and the gateway responds with allow, step-up (HTTP 401)
or block (HTTP 403) before the back-end ever sees the request. The concrete contributions of this work are the following.

- A reference implementation of an external Zero Trust gateway in Node.js that combines JWKS-based JWT verification, runtime policy-as-code, and short-lived internal token translation, with a clear trust boundary maintained through a shared gateway secret.
- A continuous behavioural telemetry pipeline that extracts a stable, capped feature set per user and tenant over 60 s / 10 min / 60 min sliding windows, persists it in MongoDB, and recomputes tenant-level baselines on a scheduled job.
- A two-tier risk evaluation strategy: a fast, deterministic deviation-based score in the request path, paired with an asynchronous Isolation Forest service that consumes the same feature stream over RabbitMQ and records anomaly labels for offline review.
- A working, test-covered evaluation that exercises cold-start handling, policy-driven enforcement, multi- tenant isolation and the gateway-as-sole-authority guarantee, including the MISSING_GATEWAY_IDENTITY rejection path when the back-end is reached directly.

Section 2 places ZTaaS in the context of recent Zero Trust literature. Section 3 walks through the architecture and the request lifecycle, including all five diagrams referenced from the body. Section 4 covers the methodology— feature extraction, the deviation score, and the Isolation Forest configuration. Section 5 describes how the system is built and wired in practice. Section 6 reports end-to-end behaviour. Sections 7 and 8 discuss what we learned and where the work goes next.

2 Related Work

Recent work on Zero Trust spans behavioural biometrics, trust-scoring frameworks, blockchain-anchored identity and ML-driven anomaly detection [1, 5, 8]. Nagarajan et al. propose an AI-based zero-trust pipeline for the consumer industry that fuses smartphone-sensor behavioural biometrics with a Bayesian trust score, enabling grant/deny/quarantine decisions at the workload boundary [1]. Zhang et al. apply Dirichlet-based dynamic trust evaluation to federated-learning clients to mitigate betrayal behaviours, casting aggregation as a min-max optimisation problem [2]. Rivera et al. integrate blockchain-anchored zero-knowledge proofs with multi-factor authentication, replacing centralised identity stores with a distributed verifier network [3].

On the resilience and detection axis, Ahn et al. integrate Zero Trust principles with the MITRE ATT&CK matrix to derive measurable resilience indices for phishing, ransomware and APT scenarios [4]. Sasada et al. propose web-biometrics for browser-based behavioural verification with sub-130 ms response times, a useful target for SaaS gateways [5]. Stodt et al. describe a context-aware dynamic Zero Trust architecture for IIoT that combines threat assessment with fuzzy logic for real-time permission adjustment [6]. ZEBRA introduces a Ring Oscillator PUF combined with blockchain to provide hardware-anchored device identity and tamper-proof access logging [7]. UCAP, by Lee et al., proposes an unconscious continuous authentication protocol based on keystroke dynamics, with simultaneous user-and-device verification and dynamic trust evaluation [8].

What separates ZTaaS from this body of work is mostly form factor and pragmatism. The whole platform is delivered as a drop-in reverse proxy: any HTTP back-end can be slipped behind it without code changes, which keeps the security layer genuinely application-independent. The risk pipeline is split deliberately — a deterministic, very cheap score on the request path, plus a heavier Isolation Forest pass that runs off a queue—so model freshness and availability never become a tax on user-facing

latency. And tenancy is wired into every store: policies, baselines and risk thresholds are all keyed by tenantId, with a runtime admin surface (REST + a small React UI) for changing them without redeploying.

3 System Architecture

Figure 1 sketches how the four runtime components fit together. The gateway is the only piece that talks to MongoDB and RabbitMQ. The back-end never opens a connection to either; if MongoDB is down, the gateway fails open on risk computation (see Section 5.4) and the back-end is none the wiser. The admin UI is a small React application that drives the gateway's /admin/* endpoints— it is convenient, but the system is fully operable through curl alone.

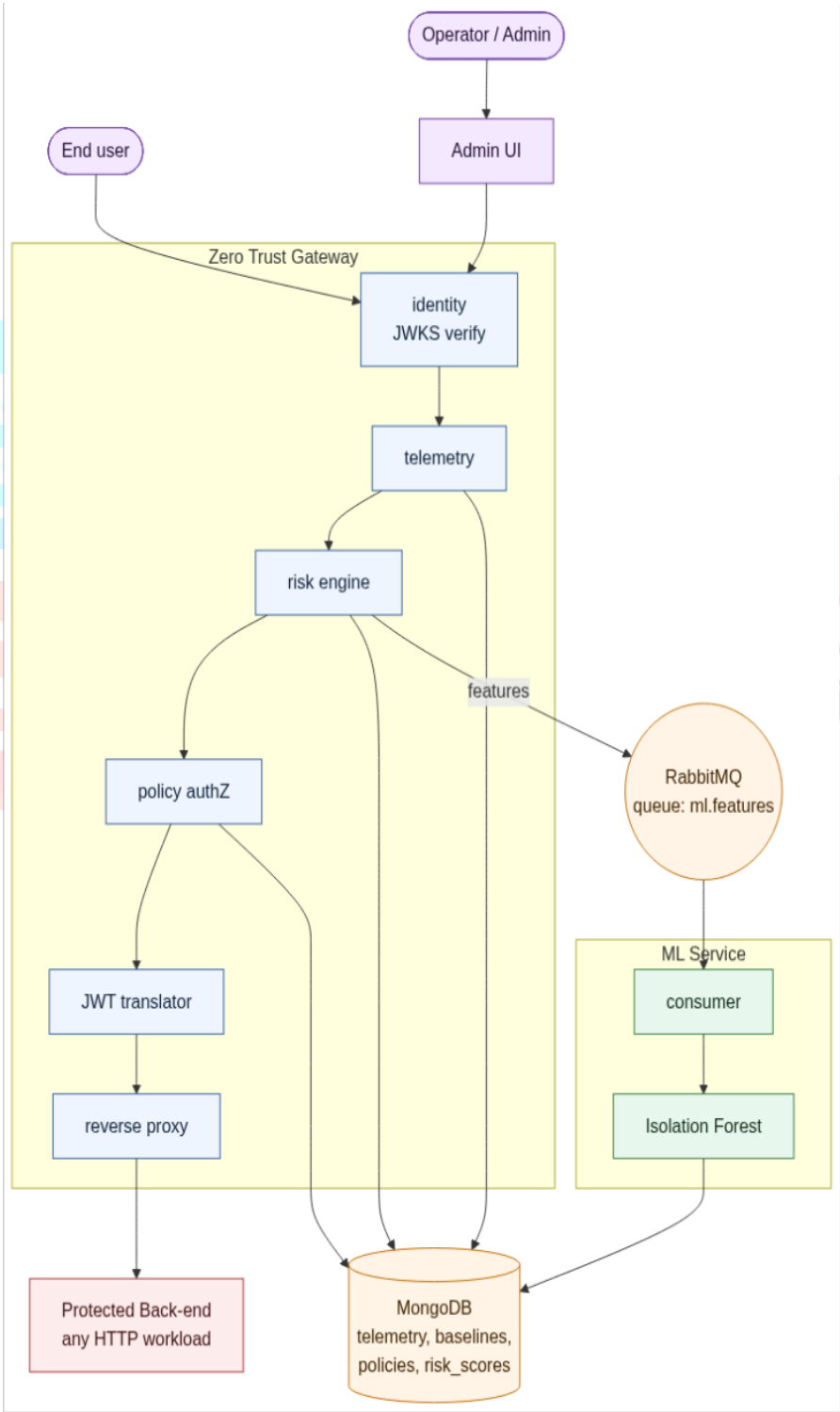

Fig. 1. ZTaaS high-level architecture. The gateway sits in front of the protected back-end and is the only platform component that talks to the data plane (MongoDB + RabbitMQ). The ML service runs off the request path.

3.1 Component Overview

Table 1. ZTaaS platform components and their responsibilities. The protected back-end is intentionally not listed: ZTaaS sits in front of any HTTP back-end and the bundled sample service is only used for local testing.

Component	Responsibility
Admin UI	Manage policies, JWT configuration, baselines and risk thresholds
Gateway	Authentication, authorisation, telemetry, risk scoring, JWT translation, reverse proxy
ML Service	Consume features from the queue, score anomalies, persist results
MongoDB	Telemetry, policies, baselines, risk_scores
RabbitMQ	Asynchronous feature stream (queue: ml.features)

3.2 Request Lifecycle

Figure 2 traces what happens between the inbound TCP connection and the outbound proxy call. The middleware chain is wired once, in proxy.routes.js, and each stage extends req with the artefacts that the next one needs:

```
router.all('*',
identityMiddleware,          // 1. JWKS verify, attach req.identity telemetryMiddleware,          // 2. record per-request features riskMiddleware,                // 3. compute score, publish, maybe block
authorizationMiddleware,    // 4. evaluate policy-as-code jwtTranslationMiddleware,  // 5. mint short-lived internal JWT handleProxyRequest          // 6. forward to back-end
);
```

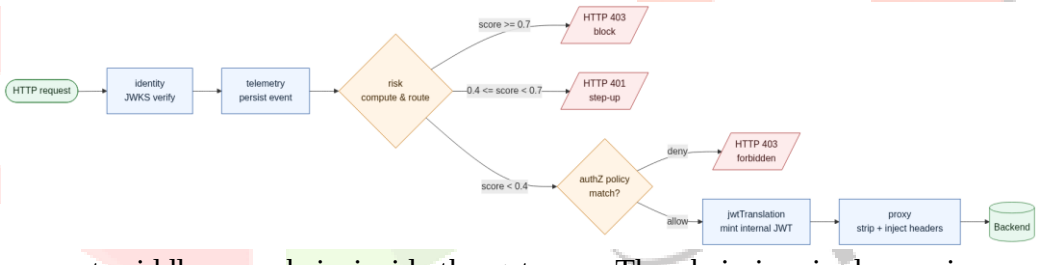

Fig. 2. Per-request middleware chain inside the gateway. The chain is wired once in proxy.routes.js; each stage reads or extends req with the artefacts the next stage needs.

Stage 1 verifies the external JWT against a JWKS cached for ten minutes and rate-limited to ten fetches per minute. Stage 2 writes a telemetry document for the request—endpoint, method, IP, user agent, latency and status—and is what feeds every downstream analytical query. Stage 3 is the operational heart of the system: it reads the user's recent behaviour out of MongoDB, computes a deviation score against the baseline, publishes the same feature vector to the ml.features queue, and may short-circuit the chain with a 401 or 403 before any further processing happens. Stages 4 and 5 then perform the policy decision and translate the request to an internal RS256-signed token that lives for one minute. Stage 6 forwards everything to the back-end with client- controlled X-User-* headers stripped and the trusted versions injected.

3.3 Trust Boundary and Defence in Depth

Figure 5 captures the trust model as a layered diagram. The back-end is configured to refuse to start unless AUTHZ_SOURCE=gateway, and on every request it checks a shared gateway secret before reading the X-User- * headers. Identity headers from clients are never trusted; the proxy strips them before forwarding. JWT validation can run in audit or enforce mode at the back-end for additional defence in depth, but the back-end takes no authorisation decision on its own. The phrase we used internally during code review captures the model:“ifthis request reached the back-end, the gateway already authorised it.”

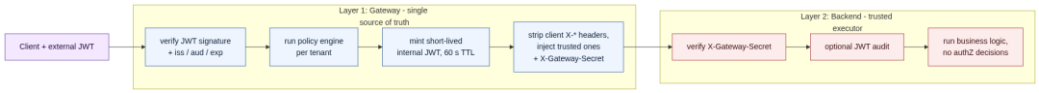

Fig. 5. Defence-in-depth view of the trust boundary. The gateway is the single source of truth for both authentication and authorisation; the back-end runs as a trusted executor.

3.4 Asynchronous ML Pipeline

The pipeline that takes features from the gateway through the model and into the analytical store is shown in Figure 3. The client sees the in-line decision before the message is even published, which matters: the queue.service publishes best-effort and silently no-ops if the channel is not yet ready. If RabbitMQ is down for ten minutes, the gateway logs the outage, gives up reconnect attempts, and continues to serve traffic with the deterministic deviation score. Nothing about the request path depends on the model being healthy.

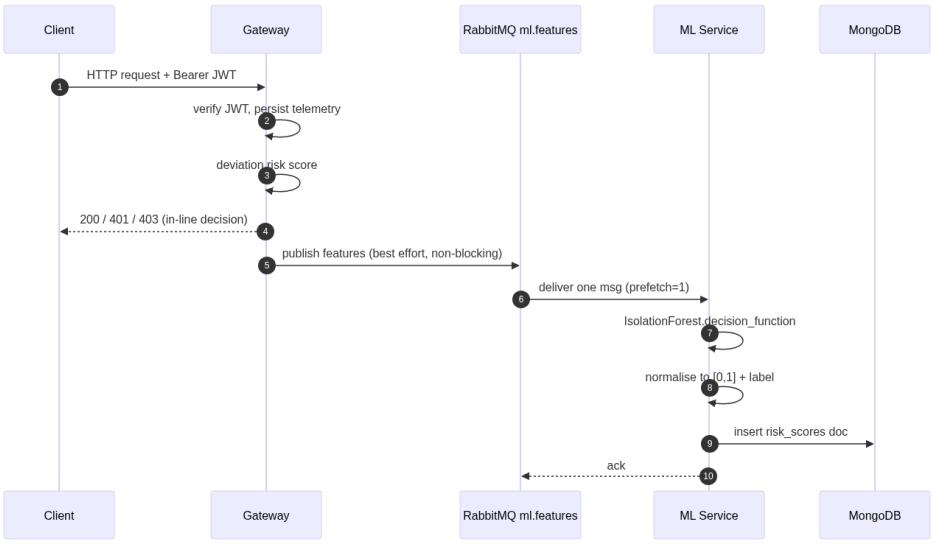

Fig. 3. Sequence diagram of the asynchronous ML pipeline. The client-facing decision is taken before the message is published; ML scoring never blocks the request path.

4 Methodology

4.1 Behavioural Feature Extraction

Each authenticated request triggers an aggregation against the telemetry collection. The query is one MongoDB aggregation that computes counts, set sizes, an average, and— slightly less common— a peak request burst inside any five-second sub-window, computed with a single linear-time sliding pointer rather than a second pass. Every numeric output is bounded by a stability cap so that a single bad five-second window cannot pull a baseline off into space:

Table 2. Per-user behavioural features computed by the gateway.

Feature	Definition	Cap
requestsPerMin	Total requests in the window	1000
failureRate	failedRequests / requestsPerMin	1.0
uniqueIPs	Distinct source IP addresses	50
uniqueUserAgents	Distinct user-agent strings	50
uniqueEndpoints	Distinct path entries accessed	200
avgResponseTime	Mean upstream latency (ms)	30 000
authDeniedCount	Authorisation denials in window	500
peakRequestBurst	Max requests in a 5 s sub-window	500

writeRatio	POST / PUT / DELETE / PATCH share of total	1.0

4.2 Multi-Window Cold-Start Mitigation

A user who logged in three seconds ago has zero requests in the last sixty seconds. Naively scoring such a user would generate a flood of 401s on every dashboard load. The gateway therefore queries three expanding windows in order — 60 s, 10 min, 60 min — and keeps the first one that returns at least one request. If all three are empty, the user is flagged as cold-start, a moderate default score of 0.4 is returned, the request is allowed, and a structured log line surfaces the situation in dashboards. The fallback path is exactly what Figure 4 illustrates.

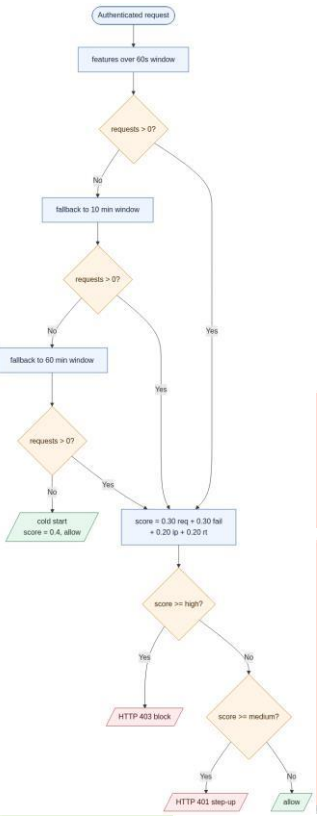

Fig. 4. Adaptive risk decision flow. The multi-window fallback is what keeps freshly authenticated users out of the false- positive bucket; the cold-start branch surfaces them in logs.

4.3 Deviation-Based Risk Score

The risk service combines four normalised deviation components against the per-tenant baseline (mean μ and standard deviation σ). A negative deviation — better than baseline — contributes zero, so users only accumulate risk by exceeding normal behaviour. The 3σ saturation pins each component to the unit interval and gives the composite score a natural [0, 1] range:

$$\text{score}(\text{value}, \mu, \sigma) = \text{clip}(\max(0, (\text{value} - \mu) / \sigma) / 3, 0, 1)$$

$$\begin{aligned} \text{risk} = {} & 0.30 \cdot \text{requestsScore} \\ & + 0.30 \cdot \text{failureScore} \\ & + 0.20 \cdot \text{ipScore} \\ & + 0.20 \cdot \text{responseTimeScore} \end{aligned}$$

The weights came out of a deliberately small calibration exercise rather than a learned process. Request-rate and failure-rate spikes turned out to be the most reliable in the scripted evaluation, so they got the larger weights; latency and IP diversity behave more like confirmatory signals and got the smaller ones. Replacing this hand- tuned mix with a learned weighting once production telemetry is available is one of the obvious follow-ups (Section 8).

4.4 Adaptive Enforcement Policy

Risk thresholds live in MongoDB on a per-tenant basis. The default is { highThreshold: 0.7, mediumThreshold:
0.4 }, constrained at write time so that medium < high. Table 3 summarises the resulting decision rule.

Table 3. Default ZTaaS adaptive enforcement decision table.

Risk score	Risk level	Action
score < 0.4	LOW	Allow
0.4≤ score < 0.7	MEDIUM	Step-up authentication (HTTP 401)
score≥ 0.7	HIGH	Block (HTTP 403)

4.5 Isolation Forest Anomaly Model

The asynchronous ML service uses an Isolation Forest [9], an unsupervised ensemble that isolates points by recursive random partitioning. Points that need fewer splits to isolate are considered more anomalous, which fits the long-tailed distribution of API behaviour very well. The model is configured with n_estimators=100, contamination=0.1 and random_state=42 for reproducibility. Until production telemetry for a tenant accumulates, the model is bootstrapped on a synthetic baseline of 18 representative samples plus three injected anomalies that calibrate the contamination boundary.

For each scoring request, the consumer evaluates the decision function, normalises the raw score into [0, 1] (1 = anomaly) and labels the event:

```
raw   = model.decision_function([requestsPerMin, failureRate, uniqueIPs, avgResponseTime])
score = clip(-raw + 0.5, 0, 1)
label = 'anomaly' if model.predict() == -1 else 'normal'
```

Each scored event is persisted with a compound (userId DESC, timestamp DESC) index so that per-user retrospectives in the dashboard remain a single seek even as the collection grows.

5 Implementation

5.1 Repository Layout

Table 4. ZTaaS repository structure.

Path	Purpose
gateway/	Express-based Zero Trust gateway (port 8081)
backend-service/	Sample protected back-end (port 5001)
admin-ui/	React + Vite admin dashboard (port 5173)
ml-service/	Python anomaly detector (RabbitMQ consumer)
docs/	Step-wise implementation notes and quick references

5.2 Gateway

The gateway boots in src/server.js — a deliberately tiny file. It connects to MongoDB, loads system configuration, kicks off the periodic baseline job (every 60 minutes, with an immediate first run so the baseline collection is never empty after deploy), and opens the RabbitMQ channel before binding the Express app on port 8081. The Express layer mounts auth, admin, JWKS, STS and proxy routers; the wildcard proxy router applies the chain from Section 3.2.

JWT verification uses jsonwebtoken with jwks-rsa, with a ten-minute key cache and a ten-fetches-per- minute rate cap so the JWKS endpoint cannot be hammered by a misconfigured gateway. The verified identity carries userId, username, role, tenant and issuer, and is attached to req.identity. The internal-JWT translator issues a one-minute RS256 token whose ctx claim carries the matched policy id,

the policy version and a UTC enforcement timestamp:

```
internalPayload = {
sub: identity.username, aud: 'backend-service',
ten: identity.tenant || 'default', ctx: {
schema_ver: '1.0.0',
decision_id: policy?.id || 'no-policy', policy_version: policy?.version || 'none', enforced_at: floor(Date.now() / 1000)
}
};
```

5.3 Telemetry Pipeline and Baseline Job

Every request lands as a document in the telemetry collection via telemetry.service.js. The baseline job (jobs/baseline.job.js) discovers all distinct tenants on each tick, then asks the baseline service to bucket the past hour into one-minute aggregates and compute mean and standard deviation for each of the four risk-engine features. The result is upserted into the baseline collection keyed by (tenantId, windowMs), so the deviation score in the request path always reads a single, precomputed document.

5.4 Risk Middleware

Most of the runtime decision logic lives in risk.middleware.js. Once it has the deviation score it fans out the same feature vector to the queue, then checks the tenant policy:

```
if (risk.riskScore >= policy.highThreshold) {
return res.status(403).json({ message: 'Access denied' });
}
if (risk.riskScore >= policy.mediumThreshold) {
return res.status(401).json({ message: 'Step-up required' });
}
next();
```

The path is fail-open by design. If the risk computation throws— typically because MongoDB is briefly unreachable— the middleware logs the failure, marks the request as fail-open, and lets it through. We chose this
over fail-closed because a transient analytics outage should never become an availability incident on the protected back-end. The trade-off is logged and shows up as a counter on the operator dashboard.

5.5 Asynchronous ML Service

The Python service is intentionally small— four files, around two hundred lines in total— to maximise reliability and keep the operational surface easy to reason about:

Table 5. Files comprising the asynchronous ML service.

File	Role
app.py	Bootstrap: warm up DB and model, then start the consumer loop
consumer.py	RabbitMQ consumer with per-message ack and reconnect retry
model.py	Isolation Forest training, scoring and normalisation
db.py	MongoDB client singleton with index management

The consumer pins basic_qos(prefetch_count=1), which means RabbitMQ never delivers a second message until the first has been acknowledged. This was a deliberate choice: a single slow scoring call should not snowball into head-of-line blocking on the queue. Connection drops trigger a five-second back-off and indefinite retry. Decoding errors and missing-feature payloads currently result in an immediate ack— a future revision will route them to a dead-letter queue rather than swallowing them silently.

5.6 Admin Surface

Operators interact with ZTaaS through both REST APIs and a small React admin UI. JWT issuer / JWKS / audience configuration, the enforcement mode (observe vs. enforce), authorisation policies, risk thresholds, baseline previews and telemetry browsing are all exposed under the /admin/* namespace, guarded by an admin identity middleware. Every configuration change takes effect immediately— there is no restart, no redeploy and no in-flight request that needs to be drained.

6 Results and Evaluation

6.1 Functional Validation

End-to-end behaviour was exercised through the test scripts shipped in the repository (test-features.sh, test-jwt- translation.sh, test-step5-4-validation.sh, test-telemetry.sh, scripts/test-risk-middleware.js). Table 6 summarises the observed outcomes.

Table 6. End-to-end functional results.

Scenario	Expected outcome	Observed
Login (alice / password123) via gateway	Backend issues RS256 JWT, gateway proxies	200 OK with accessToken
GET /orders with valid admin JWT, no policy match	Allow by default	200 OK; AUTHZ decision=allow policy=none
GET /orders with admin policy { GET: [admin] }	Allow	200 OK; AUTHZ decision=allow
POST /orders by user under policy { POST: [admin] }	Forbidden in enforce mode	403 Forbidden
Direct backend hit bypassing gateway	Reject (missing gateway identity)	401 MISSING_GATEWAY_IDENTITY
High request burst→ risk≥ 0.7	Block at gateway	403 Access denied: High risk detected
Cold-start user (no telemetry)	Allow with riskScore = 0.4	Allow; reason='No recent behavioral data'

6.2 Risk-Engine Behaviour

Table 7 illustrates representative invocations of the deviation-based risk engine against a synthetic baseline ofμ=10 req/min, σ=3, μ_failure=0.05, σ_failure=0.02. The decisions follow Table 3.

Table 7. Indicative risk-engine outputs (illustrative numbers).

Case	req/min	failure rate	uniqueIPs	avg RT (ms)	score	Decision
Normal	9	0.04	2	200	0.00	Allow
Mild burst	20	0.10	3	350	0.45	Step-up
Severe burst + failures	60	0.55	10	1500	0.92	Block
Cold start (no data)	0	0.00	0	0	0.40	Allow (cold-start flag)

These rows exercise the saturation behaviour of the score function and confirm that a high-risk request is short- circuited at the gateway— the back-end never sees it.

6.3 Operational Characteristics

The gateway adds a small, bounded amount of work to each request: a JWKS-cached signature verification, one MongoDB write for telemetry, one MongoDB aggregation for the user feature window, an in-memory policy lookup, and a single RS256 sign for the internal JWT. The RabbitMQ publish is non-blocking — the queue.service publishes best-effort and skips silently if the channel is not yet ready, so the proxied request is never delayed by the analytics fan-out. The ML service runs entirely off the request path, which means client-
observed p99 latency is independent of model throughput. In the developer setup (MongoDB, RabbitMQ and all four services running locally on a single laptop) the end-to-end latency for a /orders call was dominated by the simulated back-end's 50–100 ms artificial delay rather than by gateway overhead.

7 Discussion

A few choices in the design are worth calling out, less because they are novel and more because they shaped what the system could do. Splitting authentication from authorisation was the first one. Authentication is delegated to existing identity providers via JWTs and JWKS; the gateway never holds credentials. Authorisation, by contrast, is centralised at the gateway and expressed as JSON policies that the same evaluator feeds into the internal-JWT translator. The split keeps the trust boundary clear and lets identity providers be rotated without touching policy code.

The hybrid risk strategy was the second. A deterministic deviation score is cheap, has no model artefact and runs synchronously, which makes it appropriate for in-line enforcement. Isolation Forest, by contrast, is good at picking up subtle multivariate anomalies but introduces training, drift and freshness concerns that are awkward to manage in the request path. Running it as an asynchronous consumer over a durable queue means model unavailability or latency never shows up in user-facing requests, and we still get a richer signal for offline review and dashboards.

Multi-tenancy was wired in from day one rather than bolted on later. Policies, risk thresholds and baselines are all keyed by tenantId; the JWT translator forwards a tenant claim to the back-end; and the ML store records the tenant alongside every score. The point is that ZTaaS is a credible foundation for an actual Security-as-a- Service offering and not just a demo stack that happens to talk Zero Trust.

Limitations are honest. The Isolation Forest is currently bootstrapped on synthetic data; once a tenant accumulates production telemetry, periodic per-tenant retraining should replace the static model. The risk weighting (0.30, 0.30, 0.20, 0.20) is hand-tuned and would benefit from being learned. The ML consumer does not yet route malformed messages to a dead-letter queue. And the admin endpoints rely on a lightweight identity middleware that should be replaced with a fully audited admin authentication path before production.

8 Conclusion and Future Work

We have presented ZTaaS, an AI-driven Zero Trust Security-as-a-Service platform that combines a centralised reverse-proxy gateway, runtime policy-as-code, short-lived internal token translation, continuous behavioural telemetry and an asynchronous Isolation Forest anomaly model. The architecture demonstrates that Zero Trust principles can be delivered as an external, application-independent layer that adapts security decisions to real- time risk without modifying the protected back-end.

Future work falls into four buckets. First, scheduled per-tenant retraining of the Isolation Forest, with model versioning so that an unsafe update can be rolled back. Second, a learned risk weighting that replaces the hand- tuned (0.30, 0.30, 0.20, 0.20) split. Third, dead-letter queue support in the ML consumer and distributed-trace propagation across the gateway/back-end boundary so that anomaly events carry the request context that produced them. Fourth, integration with hardware-rooted device identity (for instance the PUF-based attestations explored in [7]) to give workload-to-workload calls a stronger trust anchor than a shared secret.

Declarations

Funding: Not applicable.

Conflict of interest: The authors declare no competing interests. Ethics approval: Not applicable.

Data availability: All synthetic data and configuration used in the experiments are included in the project repository.

Code availability: The full source code for the gateway, back-end, admin UI and ML service is available in the project repository under the same authors.

References

1. Nagarajan, S.M., Devarajan, G.G., Thangakrishnan, M.S., Ramana, T.V., Bashir, A.K., AlZubi, A.A.: Artificial Intelligence-Based Zero Trust Security Approach for Consumer Industry. IEEE Internet Things J. (early access, 2024). https://doi.org/10.1109/JIOT.2024.3500000

2. Zhang, X., Wang, D., Zhu, Y., Chen, W., Chang, Z., Han, Z.: Zero-Trust Based Robust Federated Learning Against Betrayal Behaviors. IEEE Internet Things J. (early access, 2025). https://doi.org/10.1109/JIOT.2025.11090036

3. Rivera, J.J.D., Muhammad, A., Song, W.-C.: Securing Digital Identity in the Zero Trust Architecture: A Blockchain Approach to Privacy-Focused Multi-Factor Authentication. IEEE Access 11, 123456–123467 (2023). https://doi.org/10.1109/OJCOMS.2024.3391728

4. Ahn, G., Jang, J., Choi, S., Shin, D.: Research on Improving Cyber Resilience by Integrating the Zero Trust Security Model with the MITRE ATT&CK Matrix. IEEE Access 12, 89291–89309 (2024). https://doi.org/10.1109/ACCESS.2024.3417182

5. Sasada, T., Taenaka, Y., Kadobayashi, Y., Fall, D.: Web-Biometrics for User Authenticity Verification in Zero Trust Access Control. IEEE Access 12, 129611–129622 (2024). https://doi.org/10.1109/ACCESS.2024.3413696

6. Stodt, F., Reich, C., Theoleyre, F.: Beyond Static Security: A Context-Aware and Real-Time Dynamic Zero Trust Architecture for IIoT Access Control. IEEE Internet Things J. 12(17), 35380–35393 (2025). https://doi.org/10.1109/JIOT.2025.3579028

7. Alsulami, F., Kulkarni, A.R., Hazari, N.A., Niamat, M.Y.: ZEBRA: Zero Trust Architecture Employing Blockchain Technology and ROPUF for AMI Security. IEEE Access 12, 119868–119883 (2024). https://doi.org/10.1109/ACCESS.2024.3449702

8. Lee, J.-S., Chen, T.-H., Chew, C.-J., Wang, P.-Y., Fan, Y.-Y.: Unconsciously Continuous Authentication Protocol in Zero-Trust Architecture Based on Behavioral Biometrics. IEEE Trans. Reliab. 74(2), 2591–2604 (2025). https://doi.org/10.1109/TR.2025.10937066

9. Liu, F.T., Ting, K.M., Zhou, Z.-H.: Isolation Forest. In: Proc. IEEE International Conference on Data Mining (ICDM), pp. 413–422 (2008). https://doi.org/10.1109/ICDM.2008.17

10. Rose, S., Borchert, O., Mitchell, S., Connelly, S.: Zero Trust Architecture. NIST Special Publication 800-207, National Institute of Standards and Technology (2020). https://doi.org/10.6028/NIST.SP.800-207

Appendix A Mermaid sources for figures

All architecture figures in this paper are generated from Mermaid diagram sources kept under version control alongside the build script, so they can be re-rendered or edited without leaving the repository. The sources are reproduced here for completeness.

Fig. 1 — System architecture

```
graph TD
classDef edge fill:#eef5ff,stroke:#2c5aa0,stroke-width:1px,color:#0b1d33 classDef store
fill:#fff4e6,stroke:#c46c00,stroke-width:1px,color:#3d2300 classDef ml
fill:#e8f7ee,stroke:#2f7d3a,stroke-width:1px,color:#0d2912 classDef be
fill:#fdecec,stroke:#a83232,stroke-width:1px,color:#3a0e0e classDef ui
fill:#f3e8ff,stroke:#6a3aa3,stroke-width:1px,color:#22082f

U([End user]):::ui
A([Operator / Admin]):::ui UI[Admin UI]:::ui

subgraph EDGE["Zero Trust Gateway"]
ID[identity<br/>JWKS verify]:::edge TM[telemetry]:::edge
RK[risk engine]:::edge AZ[policy authZ]:::edge
TR[JWT translator]:::edge PX[reverse proxy]:::edge
end

DB[(MongoDB<br/>telemetry, baselines,<br/>policies, risk_scores)]:::store



MQ((RabbitMQ<br/>queue: ml.features)):::store

BE["Protected Back-end<br/>any HTTP workload"]:::be subgraph MLBOX["ML Service"]
CO[consumer]:::ml
IF[Isolation Forest]:::ml end

U --> ID
A --> UI --> ID
ID --> TM --> RK --> AZ --> TR --> PX --> BE TM --> DB
RK --> DB AZ --> DB
RK -- features --> MQ
MQ --> CO --> IF --> DB
```

Fig. 2 — Gateway middleware chain

```
flowchart LR
classDef step fill:#eef5ff,stroke:#2c5aa0,color:#0b1d33 classDef decide
fill:#fff4e6,stroke:#c46c00,color:#3d2300 classDef stop fill:#fdecec,stroke:#a83232,color:#3a0e0e
classDef pass fill:#e8f7ee,stroke:#2f7d3a,color:#0d2912

REQ([HTTP request]):::pass --> M1[identity<br/>JWKS verify]:::step M1 --> M2[telemetry<br/>persist
event]:::step
M2 --> M3{risk<br/>compute & route}:::decide
M3 -- score >= 0.7 --> B1[/HTTP 403<br/>block/]:::stop
M3 -- 0.4 <= score < 0.7 --> B2[/HTTP 401<br/>step-up/]:::stop M3 -- score < 0.4 --> M4{authZ
policy<br/>match?}:::decide
M4 -- deny --> B3[/HTTP 403<br/>forbidden/]:::stop
M4 -- allow --> M5[jwtTranslation<br/>mint internal JWT]:::step M5 --> M6[proxy<br/>strip + inject
headers]:::step
M6 --> BE[(Backend)]:::pass
```

Fig. 3 — Asynchronous ML pipeline

```
sequenceDiagram autonumber
participant Cl as Client participant GW as Gateway
participant MQ as RabbitMQ ml.features participant ML as ML Service participant DB as MongoDB

Cl->>GW: HTTP request + Bearer JWT
GW->>GW: verify JWT, persist telemetry GW->>GW: deviation risk score
GW-->>Cl: 200 / 401 / 403 (in-line decision)
GW->>MQ: publish features (best effort, non-blocking) MQ->>ML: deliver one msg (prefetch=1)
ML->>ML: IsolationForest.decision_function ML->>ML: normalise to [0,1] + label
ML->>DB: insert risk_scores doc ML-->>MQ: ack
```

Fig. 4 — Adaptive risk decision flow

```
flowchart TD
classDef step fill:#eef5ff,stroke:#2c5aa0,color:#0b1d33 classDef decide
fill:#fff4e6,stroke:#c46c00,color:#3d2300 classDef out fill:#e8f7ee,stroke:#2f7d3a,color:#0d2912
classDef block fill:#fdecec,stroke:#a83232,color:#3a0e0e

A([Authenticated request]):::step --> B[features over 60s window]:::step B --> C{requests > 0?}:::decide
C -- No --> D[fallback to 10 min window]:::step D --> E{requests > 0?}:::decide
E -- No --> F[fallback to 60 min window]:::step F --> G{requests > 0?}:::decide
G -- No --> H[/cold start<br/>score = 0.4, allow/]:::out G -- Yes --> I
E -- Yes --> I
C -- Yes --> I[score = 0.30 req + 0.30 fail<br/>+ 0.20 ip + 0.20 rt]:::step I --> J{score >= high?}:::decide
J -- Yes --> K[/HTTP 403 block/]:::block J -- No --> L{score >= medium?}:::decide
L -- Yes --> M[/HTTP 401 step-up/]:::block L -- No --> N[/allow/]:::out
```

Fig. 5 — Defence-in-depth trust boundary

```
graph LR
classDef ext fill:#f3e8ff,stroke:#6a3aa3,color:#22082f classDef gw
fill:#eef5ff,stroke:#2c5aa0,color:#0b1d33 classDef be fill:#fdecec,stroke:#a83232,color:#3a0e0e

C["Client + external JWT"]:::ext

subgraph L1["Layer 1: Gateway - single source of truth"] L1A["verify JWT signature<br/>+ iss / aud /
exp"]:::gw L1B["run policy engine<br/>per tenant"]:::gw
L1C["mint short-lived<br/>internal JWT, 60 s TTL"]:::gw
L1D["strip client X-* headers,<br/>inject trusted ones<br/>+ X-Gateway- Secret"]:::gw
end

subgraph L2["Layer 2: Backend - trusted executor"] L2A["verify X-Gateway-Secret"]:::be
L2B["optional JWT audit"]:::be
L2C["run business logic,<br/>no authZ decisions"]:::be end

C --> L1A --> L1B --> L1C --> L1D --> L2A --> L2B --> L2C
```