International Journal of Creative and Open Research in Engineering and Management
ISSN: 3108-1754 (Online)
Volume 02 Issue 11 November-2026 | Impact Factor: 3.5



Design and Implementation of a Zero Trust API Gateway for Secure and Risk-Aware API Access

Angshuman Banerjee[1], Arshadali M. Athani[2], Basappa Veerappa[3], Mohammed Shazaan Aarish[4], Dr. Annie Sujith[5]

[1],[2],[3],[4] Department of Computer Science and Engineering, RV Institute of Technology and Management, Bengaluru, Karnataka, India

[5] Associate Professor and Project Guide, Department of Computer Science and Engineering, RV Institute of Technology and Management, Bengaluru, Karnataka, India

Corresponding author: mohammedshazan85@gmail.com | ORCID: https://orcid.org/0009-0002-8419-5120

How to Cite this Article:

Banerjee, A., Athani, A. M., Veerappa, B. & Aarish, M. S. (2026). Design and Implementation of a Zero Trust API Gateway for Secure and Risk-Aware API Access. International Journal of Creative and Open Research in Engineering and Management, 02(11), 1-9. https://doi.org/10.55041/ijcope.v2i11.027

License:

This article is published under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

© The Author(s). Published by International Journal of Creative and Open Research in Engineering and Management.





https://doi.org/10.55041/ijcope.v2i11.027

Abstract—

APIs connect clients, applications, and services, but perimeter-based controls alone do not provide sufficient protection when credentials are compromised or traffic crosses distributed environments. This paper describes the design and implementation of a Zero Trust API gateway developed as an inline policy-enforcement layer for protected services. The prototype is implemented with FastAPI and combines password-based authentication, OAuth sign-in, JSON Web Tokens, time-based one-time-password multi-factor authentication, scoped API keys, refresh-token rotation, and role- and scope-based authorization. Its request pipeline includes IP blocking, request logging, adaptive risk scoring, sliding-window rate limiting, and a rule-based web application firewall. The gateway also provides a reverse-proxy component that resolves registered upstream services, applies destination checks, forwards permitted requests, and records request identifiers and timing information. Additional detection components evaluate behavioral signals, impossible-travel indicators, and per-user anomalies using an Isolation Forest model when the required machine-learning dependency is available. Functional verification covers successful and rejected authentication, authorization boundaries, rate-limit responses, malicious-input detection, and proxy behavior. The available project evidence supports a functional prototype assessment; it does not establish production-scale throughput, statistically measured latency overhead, or resistance to all attack variants. The paper therefore distinguishes implemented controls from deployment requirements and identifies distributed state, stronger upstream connection validation, centralized monitoring, and broader adversarial testing as priorities for future work.

Keywords— Zero Trust Architecture; API Gateway; API Security; Risk-Based Access Control; Web Application Firewall; Anomaly Detection

I. INTRODUCTION

Application programming interfaces (APIs) are the integration points through which web applications, mobile clients, and distributed services exchange data and invoke business functions. Their accessibility and reuse support modular software development, but also expose application functions to credential abuse, unauthorized access, injection attempts, and excessive resource consumption. A network perimeter does not, by itself, establish that a request from an internal host or an authenticated client should be trusted.

Zero Trust Architecture (ZTA) addresses this limitation by requiring explicit authentication and authorization, applying least-privilege access, and making access decisions with relevant context rather than relying on network location [1]. NIST describes ZTA as an approach in which subject and device access to resources is evaluated through policy-driven decisions. For API systems, this principle can be implemented at an intermediary policy-enforcement point that validates identity, evaluates permissions, inspects requests, and records security-relevant events.

This work presents a prototype API gateway intended to centralize these controls. Its contribution is an integrated implementation that combines user and machine authentication, token lifecycle management, request inspection, rate limiting, risk evaluation, anomaly



detection, and reverse proxying. The study focuses on the implemented design and functional behavior. It does not claim that the prototype has undergone a complete independent penetration test or production-scale performance evaluation.

II. RELATED WORK

NIST SP 800-207 provides the architectural basis for Zero Trust, emphasizing resource-centric protection, least privilege, and continuous evaluation of access decisions [1]. NIST SP 800-207A extends these principles to cloud-native applications and service-based environments, where identity and policy enforcement must operate across application components [2]. These publications motivate the use of an API gateway as one enforcement point, while recognizing that a gateway alone does not constitute an enterprise-wide Zero Trust architecture.

OAuth 2.0 defines an authorization framework for delegated access [3], while JSON Web Tokens provide a compact claims representation that can be signed and validated by relying parties [4]. JWT security guidance recommends careful algorithm selection, validation of claims, and protection against implementation errors [5]. The prototype uses JWTs for user sessions and separately supports scoped API keys for machine clients. Token expiry, token-version checks, and refresh-token rotation provide additional controls beyond signature verification.

The OWASP API Security Top 10 identifies risks including broken object-level authorization, broken authentication, and unrestricted resource consumption [6]. These risks show why authentication must be complemented by authorization and resource controls. Prior work on gateway-based microservice security also demonstrates the value of placing security agents near service entry points [7]. The present prototype builds on these ideas by correlating request controls with account-level risk and operational logging. Its regex-based WAF remains a heuristic layer and should not be interpreted as a replacement for a mature application security gateway.

III. SYSTEM DESIGN AND METHODOLOGY

A. Architecture and Request Processing

The gateway is implemented in Python using FastAPI. It is deployed logically between clients and registered upstream services and acts as an application-layer policy-enforcement point. The documented middleware order is: IP blocking, security headers, request logging, risk scoring, rate limiting, WAF inspection, CORS handling, session processing, and route execution. Authentication and authorization dependencies are then applied by protected routes. The order is intentional: blocked sources can be rejected early, request activity remains observable, and behavior signals can be evaluated before rate-limit rejection.



Fig. 1. High-level gateway architecture. The implementation also includes token lifecycle controls, account-risk evaluation, and optional per-user anomaly detection.

B. Authentication and Authorization

The authentication routes support account registration, password login, password reset, token refresh, and logout. Passwords are hashed with Argon2. The login flow checks account status and freeze state before issuing an access-token and refresh-token pair. JWT claims include the subject, expiration, token version, and MFA state. The token version is compared with the current user record, allowing previously issued access tokens to be invalidated when the version is incremented.

The prototype supports Google and GitHub OAuth sign-in and TOTP-based MFA. API keys are intended for machine-to-machine clients; they are generated using a cryptographically secure random source, stored as hashes with identifiable prefixes, and associated with scopes such as service-specific proxy permissions. Authorization therefore depends not only on successful authentication but also on the requested service and the principal's granted permissions.

C. Refresh-Token Lifecycle

Refresh tokens are stored as hashes and rotated when exchanged. Each successful refresh invalidates the presented token and issues a replacement within the same token family. Reuse of an already-rotated token is treated as a compromise indicator and can cause the associated family to be revoked. Password reset and logout also invalidate refresh-token state and increment the user's token version. MFA verification state is preserved during refresh so that refreshing a session does not itself elevate an unverified session to an MFA-verified one.



D. Rate Limiting and Request Inspection

The rate limiter applies endpoint-specific sliding-window limits. The configured limits include 10 requests per 60 seconds for login and registration, 5 per 60 seconds for password-reset and MFA-verification endpoints, and 120 per 60 seconds for general endpoints. Redis is preferred for shared counters; an in-memory implementation is used as a fallback. The fallback is process-local and therefore does not provide equivalent enforcement across multiple gateway workers.

The WAF examines URL paths and query strings after URL decoding, selected request headers, and JSON request bodies. Its compiled regular-expression rules look for indicators of SQL injection, cross-site scripting, path traversal, and command injection. A detected match is logged as a security event and the request is rejected. Repeated WAF hits can contribute to automatic IP blocking. This is a lightweight heuristic detector: encoding variations, parser differences, and novel payloads can evade pattern-based inspection, while benign inputs can trigger false positives.

E. Risk, Behavioral, and Anomaly Detection

The request-risk component combines authentication, behavioral, and pattern signals using the implemented weighted expression: $R = 0.30A + 0.40B + 0.30P$, where A, B, and P denote authentication, behavior, and request-pattern risk. A high behavior-risk value incurs an additional penalty, and the final score is capped at 1.0. Policy bands classify requests for allowance, monitoring, challenge, or blocking. A challenge classification is not equivalent to an automatic HTTP rejection in every route; sensitive-route step-up requirements are enforced through account-risk and MFA dependencies.

Persistent account risk is maintained separately from per-request risk. The implementation uses thresholds for step-up verification and critical account handling, with time-based decay so that risk can decrease after suspicious activity stops. Impossible-travel detection compares login context when location information is available; it is treated as a signal because VPNs, proxies, and mobile networks can produce misleading location changes.

An optional Isolation Forest component evaluates per-user feature vectors containing request frequency, failed-authentication frequency, transformed request-body size, and hour of day. The model is unsupervised and is enabled only when the relevant dependency is available and sufficient samples have been collected. Model artifacts are integrity-checked before loading. Per-user models and local persistence are suitable for a prototype but introduce memory, distribution, and operational constraints at larger scale.

F. Reverse Proxy and Upstream Protection

The reverse-proxy route forwards authorized requests to registered services. Service records in the database are the primary source of upstream configuration; an environment-based route configuration is also supported. The proxy forwards the HTTP method, path, query parameters, body, and permitted headers while removing hop-by-hop headers. It adds request identifiers and forwarding metadata, applies a configurable upstream timeout, and maps upstream timeouts and connection failures to gateway error responses.

Before connecting to an upstream, the gateway resolves the destination and checks the resulting addresses against private, loopback, link-local, and reserved address ranges, subject to the configured private-upstream option. This reduces common server-side request forgery (SSRF) risks. However, validating DNS results separately from the eventual connection can leave a DNS-rebinding or time-of-check/time-of-use gap. Production deployment should pin or validate the actual connection destination. The gateway can add HMAC-based request-signing metadata; end-to-end authenticity depends on upstream services independently verifying that signature.

IV. FUNCTIONAL EVALUATION AND DISCUSSION

The project materials document functional verification scenarios for the principal user and security workflows. The available dashboard screenshots illustrate the interface and example states; they are not a substitute for a repeatable benchmark dataset. Accordingly, the evaluation below reports the behaviors exercised by the prototype and does not infer throughput, statistical significance, or latency overhead from dashboard counters.

TABLE I. FUNCTIONAL VERIFICATION SUMMARY

Verification scenario	Expected behavior / project evidence
Valid login	Account checks pass and an access/refresh token pair is issued.
Invalid or expired JWT	Protected request is rejected with HTTP 401.
Insufficient role or API-key scope	Access is denied with HTTP 403.
Rate-limit threshold exceeded	Request is rejected with HTTP 429 and retry



	information.
Malicious request pattern	WAF records a security event and rejects the request.
Blocked source IP	Requests from an active block entry are rejected.
Upstream timeout or connection failure	Gateway returns the corresponding timeout or bad-gateway response.
Refresh-token reuse	Reuse is rejected and the affected token family can be revoked.

The functional scenarios demonstrate that the gateway has separate responses for authentication failure, authorization failure, rate-limit violations, WAF detections, and upstream errors. This separation is important because a valid identity does not imply permission to access every service, and an allowed request may still be rejected by traffic or threat policies. The prototype also records security events and request metadata to support investigation.

The current implementation has several operational limitations. First, the in-memory rate-limit fallback is not globally consistent across workers. Second, the WAF uses regular expressions rather than full protocol and application parsing. Third, per-user anomaly models are maintained locally and are not designed for large distributed deployments. Fourth, DNS validation and connection establishment are separate operations, leaving a potential destination-validation gap. Finally, the project evidence does not include a controlled load test, measured latency distribution, false-positive rate, or independently verified attack-detection rate. These measurements are required before making quantitative performance or security-effectiveness claims.

A. Repository and Continuous-Integration Evidence

The repository README previously reported 137 tests passed and 5 tests skipped, with 63.38% code coverage [9]. However, a subsequent test run performed on November 5, 2026, produced 136 passed, 5 skipped, and 1 failed test. The newer measurement is reported in this paper and takes precedence over the earlier README summary. The failure concerns a legitimate loopback upstream URL that was classified by the WAF as an SSRF attempt.

The successful GitHub Actions run recorded on September 29, 2026 (workflow run 36636728497), completed the configured linting, import-order, type-checking, security-check, Python test-matrix, and Docker image build and container health-check jobs [10]. The workflow includes Flake8, isort, mypy, Bandit, and secret-pattern checks, with test jobs configured for Python 3.10, 3.11, and 3.12. The dependency-scan step is configured to continue on error; therefore, the successful workflow status should not be interpreted as evidence that all dependency vulnerabilities were absent.

These results support the claim that the codebase has undergone automated functional and static checks. They do not establish production-scale throughput, latency percentiles, detection accuracy, false-positive or false-negative rates, or comprehensive resistance to adversarial attacks. The CI results therefore complement, rather than replace, the functional scenarios and limitations described above.

V. EXPERIMENTAL RESULTS

Three experiments were conducted on the implemented gateway: local HTTP performance, rate-limit enforcement, and web application firewall (WAF) detection. Measurements were collected on November 5, 2026. The results characterize the prototype under the stated test conditions and should not be interpreted as production-capacity or universal security claims.

A. HTTP Performance

HTTP performance was measured against the GET /health endpoint using a real TCP loopback connection to 127.0.0.1 on a Windows development machine. The gateway ran as a single process, and no external upstream service was involved. At concurrency levels of 1, 10, and 50, the gateway processed 200, 500, and 1,000 requests, respectively. The observed throughput was 576.9, 719.8, and 544.5 requests per second, while P95 latency was 2.35, 15.31, and 101.76 ms, respectively. All 1,700 requests completed successfully.

Throughput was highest at concurrency 10. At concurrency 50, throughput decreased and P95 latency increased, indicating increased queuing or scheduling overhead under higher concurrent load. These measurements exclude network latency beyond loopback, TLS termination, reverse-proxy overhead, upstream-service latency, and multi-worker deployment effects.

TABLE II. HTTP PERFORMANCE MEASUREMENTS

Concurrency	Requests	Requests/s	Median (ms)	P95 (ms)	P99 (ms)	Errors
1	200	576.9	1.66	2.35	3.07	0
10	500	719.8	12.70	15.31	55.70	0
50	1,000	544.5	91.3	101.	108.	0



			9	76	20	

B. Rate-Limit Enforcement

The default rate-limit policy was configured to allow 120 requests per client IP within a 60-second sliding window. In the experiment, 150 requests were sent from the same client identity during one window. The gateway allowed 120 requests and returned HTTP 429 for the remaining 30 requests; no other responses were observed. Thus, all requests exceeding the configured threshold were rejected in this single-process experiment.

This result demonstrates functional enforcement of the configured limit in the tested configuration. It does not establish global consistency across multiple gateway workers. The implementation uses Redis when available and an in-memory fallback otherwise; separate worker processes using the fallback maintain separate rate-limit stores.

TABLE III. RATE-LIMIT EXPERIMENT

Outcome	Count
Requests submitted	150
Allowed requests	120
HTTP 429 responses	30
Other responses	0
Allowed fraction	80.0%
Rate-limited fraction	20.0%

C. WAF Detection

The WAF was evaluated using a small labeled corpus containing eight attack payloads and eight benign requests. The attack cases represented SQL injection, cross-site scripting, path traversal, command injection, server-side request forgery, XML External Entity injection, LDAP injection, and prototype pollution. All eight attack cases were blocked with HTTP 403, and none of the eight benign requests were blocked. On this corpus, the observed true-positive and true-negative counts were both eight, with zero false positives and zero false negatives. The resulting recall and precision were each 100%, while the false-positive and false-negative rates were each 0%.

These figures are specific to the small, handcrafted corpus and must not be interpreted as general WAF accuracy. The experiment did not evaluate obfuscation, double encoding, Unicode normalization, polyglot payloads, framework-specific bypasses, multipart requests, large bodies, real traffic samples, or comparisons against OWASP CRS or WAFBench. SSRF, XXE, and prototype-pollution patterns were evaluated in request bodies or selected headers rather than URL-path scanning to avoid false positives caused by local hostnames in full request URLs.

TABLE IV. WAF DETECTION RESULTS

Metric	Observed value
Attack cases	8
Benign cases	8
True positives	8
False negatives	0
True negatives	8
False positives	0
Recall / detection rate	100.0%
Precision	100.0%
False-positive rate	0.0%
False-negative rate	0.0%

D. Automated Test Suite and Observed Regression

The automated test suite was executed using pytest on November 5, 2026. The run completed with 136 passed tests, 5 skipped tests, and 1 failed test in 18.52 seconds, with 293 warnings. The failing test was tests/e2e/test_full_flow.py::test_complete_user_journey. The failure occurred because the WAF classified a legitimate upstream URL containing http://127.0.0.1:<port> as a server-side request forgery attempt and returned HTTP 403, where the end-to-end test expected HTTP 201.

This regression is relevant to the interpretation of the WAF results: the small labeled WAF corpus produced no false positives, but the broader end-to-end workflow exposed a false-positive case involving an approved local upstream. The test suite therefore cannot be described as fully passing. The regression should be fixed and the full test suite rerun before claiming a clean automated test status.

E. Limitations of the Evaluation

The performance experiment used a lightweight health endpoint, one gateway process, and a local loopback interface. The rate-limit experiment did not verify multi-worker consistency or behavior during Redis failure. The WAF corpus contained only eight attack and eight benign cases. In addition, the test suite contains one known end-to-end failure. Further work should repeat performance conditions across multiple trials, evaluate authenticated and proxy endpoints, test distributed rate limiting, expand the WAF corpus with mutated and encoded payloads, and add regression cases for legitimate internal upstream URLs.



VI. CONCLUSION AND FUTURE WORK

This paper has described a FastAPI-based Zero Trust API gateway that consolidates identity validation, authorization, token lifecycle management, traffic control, request inspection, risk evaluation, anomaly detection, and reverse proxying. The prototype demonstrates how an API entry point can enforce multiple controls before forwarding requests to registered services. Its functional workflows cover authentication, scoped access, rate limiting, WAF rejection, IP blocking, and upstream error handling.

The implementation should be regarded as a research prototype rather than a production-ready security product. Future work will include deployment with PostgreSQL and highly available Redis, centralized event collection and alerting, a mature WAF engine, stronger upstream destination pinning, formalized database migrations, and distributed model inference. A reproducible evaluation should measure latency percentiles, throughput under concurrent load, resource use, false-positive and false-negative rates, and behavior under adversarial payloads. These experiments would provide a stronger basis for comparing the gateway with baseline proxy configurations.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering, RV Institute of Technology and Management, Bengaluru, for providing the facilities and institutional support for this project. The authors also thank Dr. Annie Sujith for her guidance and feedback during the design, implementation, and documentation of the work.

REFERENCES

[1] S. W. Rose, O. Borchert, S. Mitchell, and S. Connelly, “Zero Trust Architecture,” NIST Special Publication 800-207, National Institute of Standards and Technology, 2020. doi: 10.6028/NIST.SP.800-207.

[2] R. Chandramouli and Z. Butcher, “A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Location Environments,” NIST Special Publication 800-207A, National Institute of Standards and Technology, 2023. doi: 10.6028/NIST.SP.800-207A.

[3] D. Hardt, “The OAuth 2.0 Authorization Framework,” RFC 6749, Internet Engineering Task Force, 2012. doi: 10.17487/RFC6749.

[4] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, Internet Engineering Task Force, 2015. doi: 10.17487/RFC7519.

[5] M. Jones, B. Campbell, and N. Sakimura, “JSON Web Token Best Current Practices,” RFC 8725, Internet Engineering Task Force, 2020. doi: 10.17487/RFC8725.

[6] OWASP Foundation, “OWASP API Security Top 10—2023,” OWASP Foundation, 2023.

[7] R. Xu, W. Jin, and D. Kim, “Microservice Security Agent Based on API Gateway in Edge Computing,” Sensors, vol. 19, no. 22, Art. no. 4905, 2019. doi: 10.3390/s19224905.

[8] M. Jones and D. Recordon, “The OAuth 2.0 Authorization Framework: Bearer Token Usage,” RFC 6750, Internet Engineering Task Force, 2012. doi: 10.17487/RFC6750.

[9] Arshadali04, “Zero Trust Secure API Gateway,” GitHub repository. [Online]. Available: https://github.com/Arshadali04/zero-trust-secure-api-gateway. Accessed: Nov. 5, 2026.

[10] Arshadali04, “CI workflow run 36636728497,” GitHub Actions, Sep. 29, 2026. [Online]. Available: https://github.com/Arshadali04/zero-trust-secure-api-gateway/actions/runs/36636728497.