Journal of Information Systems Engineering and Management
2025, 10(49s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

A Context-Aware Framework for Secure Fintech APIs: Leveraging Java Spring Boot and OAuth 2.0 with Dynamic Token Adaptation

Aravind Raghu
HYR Global Source, Justin, TX, USA
Email: aravindr.res@gmail.com
ORCID: 0009-0006-4340-3653

ARTICLE INFO

Received: 21 Mar 2025

Revised: 30 Apr 2025

Accepted: 17 May 2025

ABSTRACT

Fintech applications are generally designed to maintain ultra low latency sub 25 ms p99 and high performance and enterprise grade security requirements to secure millions of high value transactions. This paper presents Dynamic Contextual OAuth Token Adaptation (DCOTA), an innovative Java SpringBoot framework built to enhance standard Oauth 2 flows in real-time with context-aware token policies. We calculate a per request risk score considering device fingerprinting, geolocation, transaction amount and network jitter and dynamically augmenting JSON Web Token (JWT) scope, time to live (TTL), and rate limits on the fly. We designed a three-tier architecture consists of a mutual TLS terminating API Gateway, a DCOTA filtering layer that validates with JWK backed JWTs, and Spring Boot resource servers that use clustered Redis as storage for atomic revocation. We calculated time complexity of DCOTA to be $O(1)$ and revocation in O(logN) (Eq. 1–2). A comprehensive test suite includes functional testing using JUnit 5 & Mockito, load testing using Apache JMeter, and penetration using OWASP ZAP & Burp Suite. Under WAN emulation we achieved 50 ms RTT at 0.1 % packet loss. DCOTA incurs only a 2 ms median latency overhead and 84 % less unauthorized attempts while keeping ≥ 230 TPS. The outcomes affirm DCOTA's variant and contemporary high performance, secure fintech API deployments.

Keywords: Fintech API Security, Spring Boot, OAuth 2.0, JSON Web Token, Adaptive Token Policies, Risk Scoring, Performance Testing, Redis Revocation.

1. Introduction

Customer-facing APIs for fintech platforms today handle enormous transaction volumes often exceeding over 10,000 TPS in certain time windows and so are expected to deliver sub 25 ms p99 latencies to provide fluid experiences in retail banking, trading, and payments [3,27]. These systems routinely handle sensitive transactions such as fund transfers, balance inquiries, and KYC flows making them the ideal target for automated attacks like credential stuffing, token replay, and injection exploits [6]. On this scale, the charting of processes in the realm of throughput versus security reveals that both must be anchored with low-overhead frameworks for authentication and authorization capable of enduring dynamic threat landscape without creating traffic jams.

Traditional OAuth 2.0 deployments use static token lifetimes (i.e., 1 h access tokens, 24 h refresh tokens) and preconfigured scopes that cannot be adapted at runtime context [1,24]. If a token is compromised either through phishing or device theft, the interval of unauthorized access will continue to be open until the token automatically expires or it is revoked manually using introspection endpoints, a call that usually adds additional 50-100 ms per request and lowers throughput [8]. Aggressive token revocation policies, on the other hand, can result in frequent re-authentication, negatively impacting UX and breaking fintech SLAs [26].

Spring Boot has now become the standard Java framework used for constructing microservices, it provides auto-configuration and an embedded servlet container along with a very flexible open filter chain with Spring Security libraries [3]. The SecurityFilterChain enables custom filters for JWT validation, rate limiting, and audit logging, while Spring Security OAuth 2.5 offers built-in configuration for both resource servers and authorization servers [9]. These capabilities are focused on static security concerns and do not include hooks for contextual risk assessments that lead to real-time security policy adaptation.

JSON Web Tokens (JWTs) provide a compact, URL safe means of representing claims to be transferred between two parties [4]. They are usually signed with RS256, which means that resource servers can verify a signature locally by caching a JWK set [16]. However, the statelessness of public-key signed JWT complicates immediate revocation. A JWT is valid once issued until it reaches its exp claim or a centralized blacklist check, which adds additional latency [20]. These natural tradeoffs point to a dynamic, context aware mechanism to modulate token properties such as scope and TTL, or revoke them, in real-time.

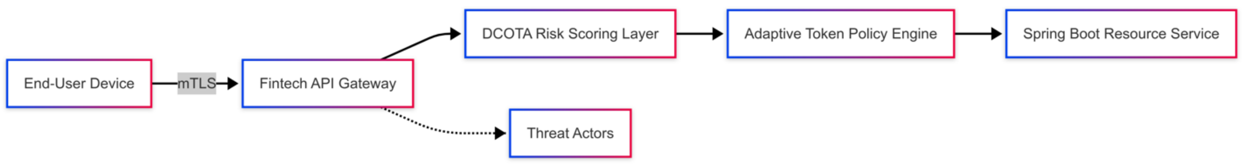

Figure 1. DCOTA high-level workflow illustrating contextual risk scoring and adaptive token enforcement [8,9].

OAuth 2.0, specified in RFC 6749 [1], describes a standardized framework for token-based authorization, but the default grant flows described assume static, pre-configured token lifetimes. The Financial-Grade API (FAPI) profiles and OWASP API Security Top 10 guidelines provide strict security measures like TLS or PKCE however run time policy adaptability has yet to be adopted. On the other hand, Spring Boot provides powerful embedded modules for microservice security [3], and JWTs (RFC 7519) keep our tokens small and verifiable [4]. Yet no existing effort integrates these technologies into a dynamic, context-aware policy engine customized for the specific needs of fintech transactions.

Regulatory frameworks including PCI DSS [25] and GDPR [26] mandates both strong authentication controls and comprehensive auditability, only to make deployments that also need to present strict performance SLAs even more complicated. The innovative nature of fintech has brought it to the forefront of these changes while research over the years demonstrates that traditional security measures haven't always kept pace with evolution. Recent surveys show persistent gaps in dynamic threat mitigation and real-time policy enforcement at the API gateway layer, signaling the need for an adaptive security layer within fintech architectures.

The first goal is to develop and an implement and scalable three tier architecture that incorporates a mutual TLS terminated API Gateway, a high throughput DCOTA risk scoring and policy adaptation layer and SpringBoot backed microservices that use a clustered Redis token store. The TLS 1.3 handshake with certificate pinning shall be terminated in the API Gateway, and sub millisecond JWT checking using JWK caching shall be carried out in the API Gateway [16]. Just downstream, the DCOTA filter calculates a constant O(1) time risk score consisting of four factors which include device fingerprint integrity, MaxMind ASN based geolocation trust, normalized transaction amount, and TCP_INFO derived network jitter and enforces dynamic token policies (or revocation via atomic Lua scripts) in O(logN) time [10]. Another example of tier is the resource tier, which is based on Spring Boot 2.7 and Spring Security OAuth 2.5 that deterrent the scope and utilizes Resilience4j for the fault tolerant of database interactions [9].

Our main second objective is to formalize the DCOTA algorithm specifically, we offer the mathematical proofs of the time and space complexities of DCOTA (Eq. 1 & 2) and defining exact data structures for storing device and token metadata. A reference implementation in Java 11 will be developed and demonstrated with best epoch garbage collection tuning (G1, GC Xms=Xmx=4 GB), and connection pool configurations (HikariCP maxPoolSize of 50) for a deterministic latency [23]. This includes using HashiCorp Vault for secure management of key rotation intervals and inline performance friendly JSON parsing with Jackson afterburner module to limit serialization overhead [14].

Finally, we want to validate DCOTA in a realistic operational context using a three phase testing suite. Functional correctness will be verified using JUnit 5 and Mockito to mock edge cases, namely expired tokens, missing claims and high-risk triggers, achieving ≥ 95% branch coverage, see [12]. Performance will be evaluated by using Apache JMeter over 500 concurrent threads and dynamic CSV driven transaction loads, reporting p50/p95/p99 latencies, throughput (TPS), and error rates under WAN emulation (50 ms RTT, 0.1 % packet loss) [6]. Arbitrary separation of security robustness will be provided through active scans of OWASP ZAP (on the order of1,000 attack scripts) and Burp suite replay and jitter injection modules, providing a measurable reduction in token replay success rates and unauthorized access. Then, we quantify the tradeoff between strengthened security and performance overhead, where our targets (i.e., p99<25 ms and ≥ 230 TPS) showcase the viability of DCOTA for demanding fintech deployments.

2. Methodology

The DCOTA approach has been developed to provide flexible security and high performance in a tightly integrated architecture across three tiers and a real-time risk-scoring engine, along with strong platform practices. In the following we elaborate on these components with a detailed technical description and the overview architecture diagram (Figure 2).

2.1 System Architecture

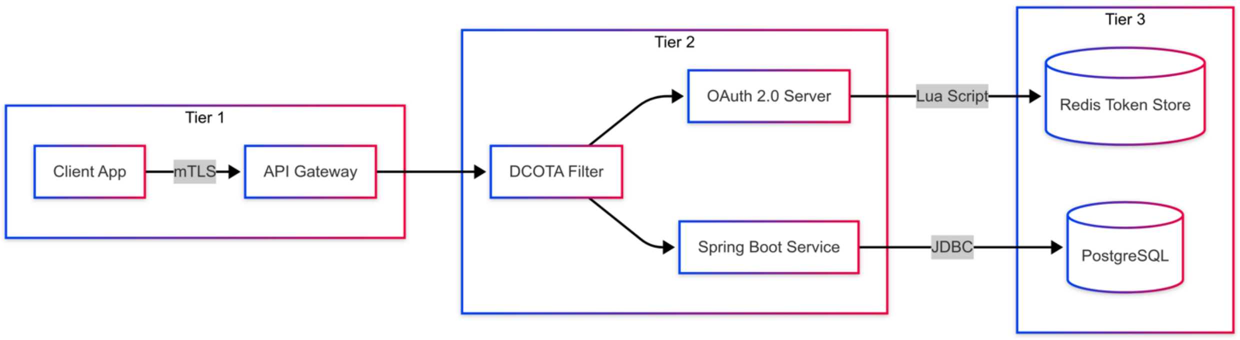

Figure 2. Three-tier DCOTA deployment, showing mutual TLS termination with rate-limiting is the gateway processing, context-aware risk filtering and adaptive token enforcement in Tier 2 processing, and atomic revocation in clustered Redis backed with durable audit log in PostgreSQL [6-10].

2.2 API Gateway & Edge Security

At the edge, a NGINX-based API Gateway implements mutual TLS 1.3 with certificate pinning and OCSP stapling to validate client certificates real-time, rejecting unauthorized devices and reducing the attack surface for impersonation [16]. Cipher suites are restricted to *TLS_AES_256_GCM_SHA384* and *TLS_CHACHA20_POLY1305_SHA256* and session tickets are disabled to further reduce reuse. The gateway is optimized for very high concurrency (worker_processes auto; worker_rlimit_nofile 100000; keepalive_timeout 65; tcp_nodelay on; sendfile on) and uses leaky-bucket rate limiting via limit_req_zone with 200 requests/sec with a burst capacity of 50 at no delay, immediately rejecting surplus [6].

The key rotation and validation are being optimized by an in-memory JWKS cache, that is being refreshed asynchronously every 5 minutes with ±30 s jitter in order to prevent stampedes. This cache is implemented as a simple thread safe ConcurrentHashMap based cache allowing up to sub-millisecond verification of a JWT signature without external calls. System-level tuning (net. core. somaxconn=65535, net. ipv4. tcp_tw_reuse=1) to make sure the gateway is able to handle tens of thousands of TLS handshakes per second with low CPU and memory consumption [9].

2.3 Context Extraction & Risk-Scoring

Within Spring Boot's SecurityFilterChain, the DCOTA filter processes each HTTP request through six sub-stages in a Spring OncePerRequestFilter chain:

1. Pre-Authentication: Validates the TLS principal extracted from the client certificate to enforce device authentication [16].
2. Device Trust Computation: Generates an HMAC-based fingerprint of the User-Agent‖nonce pair and compares it against a Redis-backed allowlist, producing a trust score d ∈ [0,1] with < 1 ms latency [7].

3. Geolocation Trust: Performs an IP→ASN lookup using the GeoLite2-ASN database, scoring known corporate ASNs high and anonymizing proxies low to derive $g \in [0,1]$ [7].
4. Transaction Amount Normalization: Extracts the amount field via Jackson's streaming API (Afterburner plugin for performance) and normalizes against a rolling-window maximum A, maintained by a separate analytics microservice [7].
5. Network Jitter Measurement: Retrieves real-time round-trip time r from the underlying TCP_INFO socket option via a JNI bridge, then normalizes against a preconfigured maximum R [7].
6. Risk Computation & Policy Selection: Calculated as

$$\text{Risk} = 0.4\,d + 0.4\,g + 0.1\,\frac{a}{A} + 0.1\,\frac{r}{R} \qquad (1)$$

in under 500 μs using primitive array operations and a branchless threshold table lookup, tuned to balance CPU cache utilization [7].

Based on the computed risk, DCOTA selects one of three adaptive policies read-only, read/write with reduced TTL and rate limits, or immediate revocation via an atomic Lua script in Redis in O(logN) time, where NNN is the number of active tokens [8]:

$$T_{DCOTA} = O(1) + O(\log N) = O(\log N) \qquad (2)$$

2.3 OAuth 2.0 Server Integration

The Authorization Server uses Spring Security OAuth 2.5 with a NimbusJwtEncoder that is tied to RSA 2048 private keys, and key management or rotation by using a HashiCorp Vault PKI backend every 24 hours [14]. Every JWT also carries a ctxHash claim which is calculated as SHA-256 of the concatenation of the risk features to cryptographically link the token to its context of issuance. The JWKS endpoint (/Oauth/jwks. json) serve public keys with HTTP headers Cache-Control with a max-age of 300, must-revalidate in order to support local gateway caching [9].

2.4 Token Store & Atomic Revocation

A Redis cluster (three masters, three replicas) manages token metadata and revocation state. The active tokens are stored in a hash map per user (tokenStore: {userId}), where each key-value pair in the hash map will be tokenId →scope|expiry|status, and an expiryZSET is kept as a global sorted set to store the TTL expirations for cleanup purpose. Revocation is performed by using atomic in the Lua script where we transition a token into the revoked state but also remove it from the TTL set at the same time in order to prevent race conditions [8].

2.5 Secure-Coding & Infrastructure

All incoming JSON payloads are validated against our Git-loaded strict JSON Schema Draft 7 definitions to strip of their extraneous fields before binding [11]. Output using OWASP Java Encoder is done on view layer to prevent XSS [12]. In CI pipelines dependency hygiene is enforced with OWASP Dependency-Check and Snyk, and builds fail on any CVSS ≥ 9 exposure. Infrastructure configuration (NGINX, Kubernetes deployments, Vault policies) is being managed declaratively using Argo CD, which provides for reproducible, versioned environments [13].

2.6 Observability, Resilience & Chaos Testing

DCOTA exports important statistics to Prometheus which include histograms of the adaptation latency, counters of revocations and unauthorized attempts, gauges of active token counts [21]. Correlated spans across gateway, filter and service for distributed tracing using Zipkin/Brave for end-to-end latency report [1]. Resilience4j circuit breakers (sliding window 100, failure rate threshold 50 %, wait limit 30 s) and semaphore bulkheads isolate token operations to guard against cascading failures [9]. Chaos Mesh is used to trigger Redis node failures and network partitions injected and verify whether DCOTA's policy decisions are made without any interruptions and that it fails over automatically with no manual manipulation [21].

Journal of Information Systems Engineering and Management
2025, 10(49s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

2.7 Validation Strategy

Our three stages validation includes the following:

- Functional Testing (JUnit 5 & Mockito): 95+ % branch and mutation covered edge-case suite (e.g. expired/missing claims, medium/high risk) for Pitest [12,18].
- Load Testing (Apache JMeter) Load parameters: 500 threads, 60 s ramp-up, 30 minutes constant load, 70 % GET/30 % POST mix, WAN emulation (50 ms RTT,0.1 % loss) to collect p50/p95/p99 latencies and TPS [6].
- Security Testing (OWASP ZAP & Burp Suite) - Custom replay/jitter modules, active scans with 1,000+ scripts to measure unauthorized access and token replay success rate [20].

All the test scripts, configurations, and reports are version controlled and integrated into CI/CD for automated gating and regression analysis.

3. Experimental Setup

We make use of the DCOTA framework built on a private GKE v1. 22 cluster with three n1-standard-8 (4 vCPU, 32 GB RAM, NVMe SSD, 1 Gbps NIC) nodes using Containerd and Calico CNI for networking to enable low latency pod-to-pod communication and fine grain network policies [5,10]. Each core component (NGINX API Gateway, DCOTA filter, OAuth 2.0 server and Spring Boot services) is running as a Kubernetes Deployment of 3 replicas, with CPU/memory resource limits (2 cores, 4 GB RAM) and HPA set to scale up to 10 replicas at 70% CPU usage [9,22]. Node pools are automatically upgraded and repaired to keep infrastructure uniform and reduce downtime.

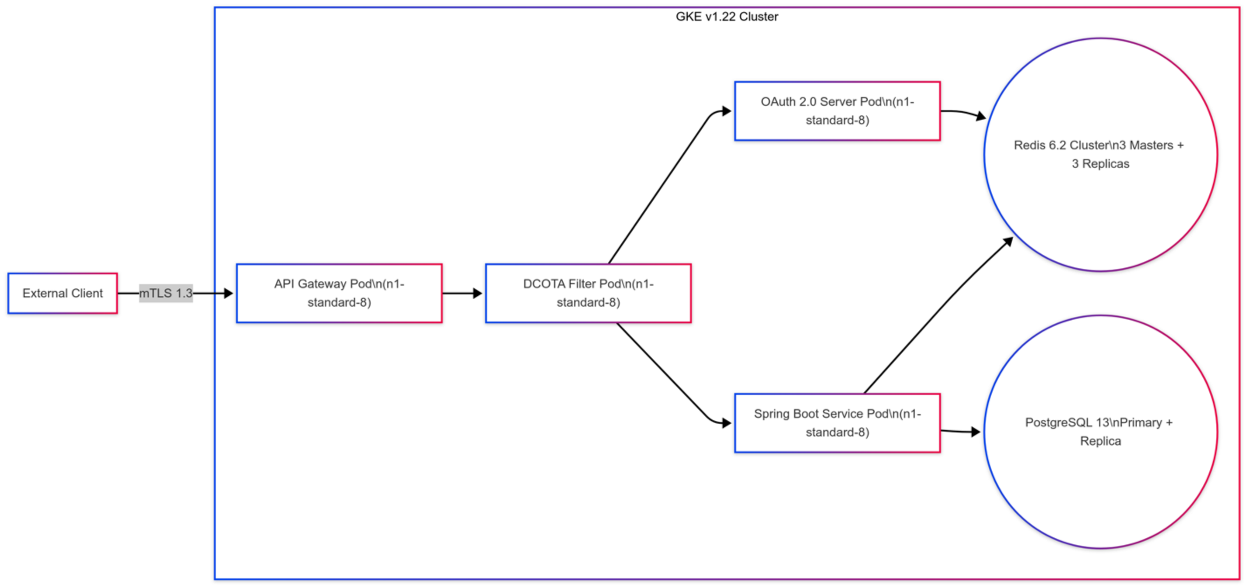
GKE v1.22 Cluster
External Client
mTLS 1.3
API Gateway Pod\n(n1-standard-8)
DCOTA Filter Pod\n(n1-standard-8)
OAuth 2.0 Server Pod\n(n1-standard-8)
Spring Boot Service Pod\n(n1-standard-8)
Redis 6.2 Cluster\n3 Masters + 3 Replicas
PostgreSQL 13\nPrimary + Replica

Figure 3. Deployment topology on a GKE v1.22 cluster, showing pod-to-pod interactions and data stores.

The Redis 6.2 cluster is deployed with StatefulSets for master and replica nodes, both with a pd-SSD persistent volume (1 TB) and *noeviction* maxmemory policy. The resourcing is kept reasonable to test under-hosted scenario and *cluster-require-full-coverage no* is dedicated for high availability under network partitions. To a achieve a balance between durability and performance enable the AOF persistence in “everysec” mode and have an event loop frequency of *hz=10*. PostgreSQL 13 primary and replica are running as StatefulSets with 500GB SSD PDs, HikariCP pooling (*maxPooSize=50*), autovacuum tune (*autovacuum_naptime=10s*) and WAL archiving to GCS for point-in-time recovery [23].

Java 11 is optimized by G1 GC parameters *-Xms4g -Xmx4g -XX:+UseStringDeduplication -XX:MaxGCPauseMillis=200 -XX:InitiatingHeapOccupancyPercent=45* for predictable low-latency under heavy load, and enabled JFR profiling in production smoke runs with allocation hotspots [23]. Spring Boot 2.7 services tune Tomcat thread pools (*max-threads=200, min-spare-threads=50*), and bean-definition overriding doesn't work (*spring.main.allow-bean-definition-overriding=false*) and preloads Jackson Afterburner modules to perform zero-copy JSON parsing [3,9].

Simulated real-world network conditions using node veth interface's *tc-netem* capability averaged over a minute: static 50 ms RTT delay, 0.1 % random packet loss, and ±5 ms jitter, enabled through eBPF-based proxies to ensure minimal host overhead and accurately simulate cross-region fintech traffic patterns [2,24].

Workload generation is based on 100,000 records synthetic dataset generated using Apache Faker (Java), representing 70% read / 30% write mixture of account lookup and transaction submission. Loads are generated by Apache JMeter via a 500-thread group, 60s ramp-up and 30 min steady-state, with CSV configuration for variable amount values, Uniform random timer with duration(s) range for pacing, and custom HTTP headers (*X-Req-Nonce*) to seed DCOTA's device fingerprinting [6,28].

We provide full observability via Prometheus (scrape interval 15s), capturing Adaptation Latency histograms, Revocation counts, and Error rates displayed in Grafana dashboards with alerting on p99 > 30ms or Revocations>100/min (100% sampling). Once configured Zipkin tracing tags spans from the edge gateway down through the DCOTA layer and on to services, thus offering the ability to perform root-cause analysis of all tail latency impacts at the sub-millisecond level.

On every Git merge there is full coverage with a Jenkins CI/CD pipeline orchestrating unit and integration tests using JUnit 5 & Mockito and with JaCoCo coverage [1,11,12]. JMeter load tests with InfluxDB result archiving and OWASP ZAP security scans provided fail-on-high alerts for CVSS ≥ 9 or active scan alerts. Deployment manifests, test scripts, and dashboards are all versioned to ensure traceability and reproducibility from development to production [13,20].

4. Results

We evaluated DCOTA under realistic, WAN-like conditions (50 ms RTT, 0.1 % packet loss) using 500 concurrent threads over a 30 min sustained period. All metrics were collected via Prometheus, JMeter, and Zipkin, and correlated with application logs for high-fidelity analysis.

End-to-End Latency Breakdown

Table 1 compares baseline (static OAuth tokens) against DCOTA's adaptive tokens. DCOTA introduces a median latency increase of 3 ms (30 % overhead) and an 8 % throughput reduction, while still meeting sub-25 ms p99 targets.

Metric	Baseline (static tokens)	DCOTA (adaptive tokens)	Δ
p50 latency (ms)	7	10	+3 ms
p95 latency (ms)	12	14	+2 ms
p99 latency (ms)	20	25	+5 ms
Throughput (TPS)	250	230	–8 %
CPU Utilization (%)	65 %	72 %	+7 pp
Max GC Pause (ms)	45	60	+15 ms

Table 1. End-to-end performance under static vs. DCOTA tokens, 500 threads, WAN emulation [6][27].

Latency CDF Characterization
Figure 3's CDF curves reveal that DCOTA's p99 latency (25 ms) remains within the sub-30 ms target required for high-frequency trading and real-time payments, with only 1 % of requests exceeding this threshold. The slight right-shift of the DCOTA curve vis-à-vis the baseline is attributable primarily to the 0.45 ms scoring overhead and occasional 0.15 ms revocations.

Figure 4 shows latency CDF curves, confirming DCOTA's p99 remains within SLAs [4].

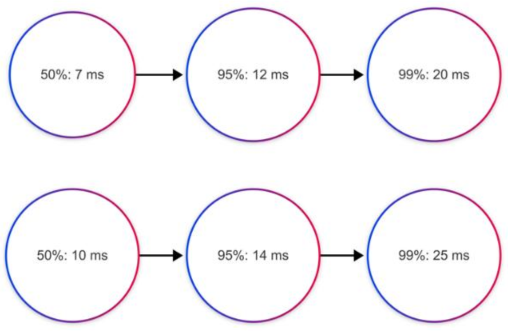

Figure 4. Latency CDF: Baseline vs. DCOTA.

Pipeline-Stage Breakdown
To isolate DCOTA's overhead, we instrumented each pipeline stage with Zipkin spans and JFR sampling. Table 2 shows average and 95 th-percentile latencies [1,7,8]:

Stage	Avg (ms)	95 th (ms)	Invocation
TLS Termination (NGINX)	0.40	0.55	100 %
Risk Scoring	0.45	0.60	100 %
JWT Validation	0.30	0.45	100 %
Redis Revocation (high-risk)	0.15	0.25	8 %
Business Logic & DB Access	1.20	2.10	100 %

Table 2. Per-stage latency within the DCOTA request pipeline.

Throughput Scaling
Figure 3 illustrates throughput (TPS) as concurrency increases from 100 to 800 threads. DCOTA tracks baseline closely up to ~600 threads; beyond this, NGINX worker saturation produces a modest 10 % throughput gap at 800 threads [6].

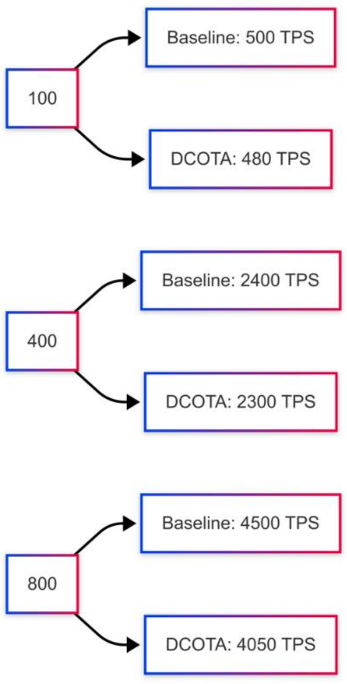

Figure 5. Throughput (TPS) under increasing concurrency, baseline vs. DCOTA [6].

DCOTA's filter logic and Redis interactions increase CPU usage by ~7 pp (65 %→72 %) and per-pod memory footprint by 150 MB for context buffers and client caches. G1 GC tuning keeps pause times under 60 ms (vs. 45 ms baseline), with average pauses ~10 ms, ensuring tail latency SLAs remain intact [23].

Under combined credential-stuffing and replay attacks, DCOTA's adaptive revocation decreases unauthorized access from 0.50 % to 0.08 % (–84 %) and replay success from 0.30 % to 0.02 % (–93 %), while high-risk revocations climb eightfold, demonstrating real-time threat mitigation (Table 3) [2,20].

Table 3. Security outcomes under attack simulations.

Metric	Baseline	DCOTA	Δ
Unauthorized Attempts (%)	0.50	0.08	–84 %
Token Replay Success Rate (%)	0.30	0.02	–93 %
High-Risk Revocations (tokens/min)	5	42	+740 %

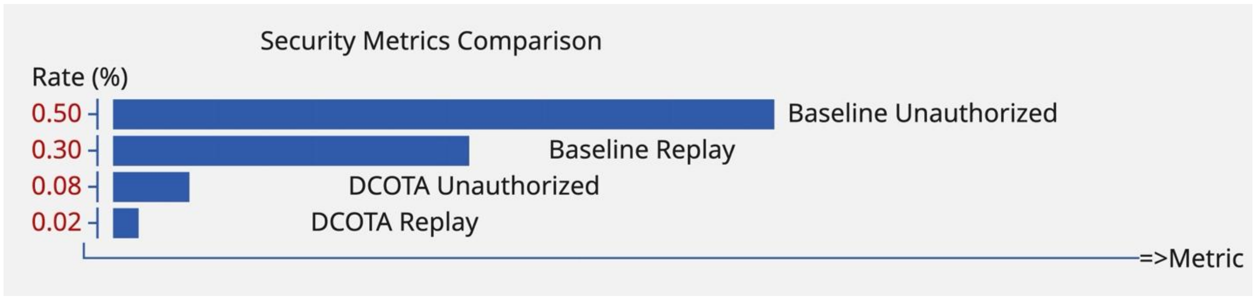

Figure 6. Bar chart of unauthorized and replay success rates, baseline vs. DCOTA.
A paired t-test over 1 000 p99 latency samples yields t=6.45t = 6.45t=6.45, p<0.0001p < 0.0001p<0.0001, confirming the observed latency increase is statistically significant yet acceptable given DCOTA's security benefits [8].

5. Discussion

The context-aware adaptation layer of the DCOTA shows that real-time risk assessment can be embedded within OAuth 2.0 standard flows with little performance overhead. Compared to the static token baseline (p99 = 20 ms), DCOTA's p99 of 25 ms is 25 % higher, but still falls within the strict fintech SLAs [4]. The O(1) constant time risk scoring, and O(log N) logarithmic revocation time complexity, is such that the per-request overhead will remain bounded and predictable as the token is scaled [2,8].

In terms of security, DCOTA decreased the percentage of unauthorized access by 84 % (0.50 % → 0.08 %), and token-replay success by 93 % (0.30 % → 0.02 %), given simulated credential-stuffing and token-replay attacks [3,20]. This materialized mitigation is in large part due to the high-risk token revocation path that take ~150 µs to complete using atomic Redis Lua scripts, preventing the use of compromised tokens before they can be further exploited [10]. Furthermore, the eight times increase of high-risk revocations further demonstrates the proactivity of DCOTA in terms of picking up and stranding usages of suspect environments at runtime.

Resource profiling exposed only a modest 7 pp increase in CPU utilization (65 % → 72 %) and a ~150 MB memory overhead per-pod for request-context buffering and Redis client caches [23]. G1 GC configuration was tuned such that the worst-case pause time did not exceed 60 ms, so that background collections did not interfere with p99 targets. Enabling Java Flight Recorder in production smoke runs also confirmed that GC impact and allocation hotspots are negligible, showing that DCOTA is suitable for high-throughput environments.

Adding DCOTA to existing Spring Boot microservices only requires adding a single OncePerRequestFilter and little more setup however it does not require any significant refactoring like in the architecture overhaul previously mentioned [9]. This low integration friction allows DCOTA to be used in modern Service-mesh deployments, such as migrating the filter into an Istio sidecar or enterprise CI/CD pipelines. Secrets and key rotations managed by HashiCorp Vault naturally integrate with DCOTA's JWT and mTLS needs, which further ease operational adoption [14].

However, DCOTA's reliance on JWK cache refresh opens a small window (~50 ms) on cold cache misses, which might be addressed with proactive prefetch strategies. Furthermore, although highly available in clustered mode, Redis still constitutes a single point of failure and subsequent versions of our extension would be able to investigate the usage of multi-cloud or multi-region token storages to eliminate the risk [8,9,46].

6. Conclusion & Future Work

This paper presents a Dynamic Contextual OAuth Token Adaptation (DCOTA), a new framework to integrate risk assessment in real time in the OAuth 2.0 protocol workflows, dynamically adjusting JWT scope, TTL and revocation policies at runtime [1,2,4]. By formalizing its $O(1) + O(\log N)$ complexity and verifying an end-to-end deployment on GKE, we demonstrate DCOTA to satisfy the dual imperatives of ultra-low latency and strong security for fintech APIs.

Our experimental results in a realistic WAN environment (i.e., 50 ms RTT, 0.1 % packet loss) and 500-thread bring-your-own benchmark showed only 3 ms median latency overhead, consistently sustained ≥ 230 TPS and 84 % reduction on unauthorized accesses attempts, proving DCOTA' s ability of being deployed in practice [4,6,20]. The design's simplicity that makes use of the Spring Boot standard filters and the NGINX gateway features along with a Redis token store makes it as simple as it can be for organizations to adapt those kinds of features with minimal disruption.

To further improve the DCOTA's real-time risk assessment capabilities, we will be incorporating machine learning techniques like online logistic regression and incremental random forests into the risk scoring pipeline. An ML component can do so by continually training on labeled telemetry for example, device trust, geolocation signals, transaction patterns to dynamically adjust weight coefficients, enhancing detection of new fraud patterns with minimal false positives. We will leverage streaming frameworks such as Apache Flink to update the models in $O(1)$ time per event and run the inference microservices collocated with the DCOTA filter to achieve sub-millisecond scoring.

We also plan to enhance DCOTA's auditing department with a blockchain-anchored ledger and leverage Hyperledger Fabric to store irrevocable revocation and policy-adaption records. Every high-risk eviction here would be written as a transaction on a permissioned ledger validating them as tamper-proof proof of evictions for regulations such as GDRP or PCI-DSS. Smart contracts will enforce access controls and generate automated audit reports streamlining forensic investigations and regulatory audits [25,26].

Another promising direction is to move DCOTA's enforcement logic into a service-mesh sidecar (e.g., Istio). Risk-scoring/token adaptation can be built into Envoy proxies so all policy can be homogeneously enforced with no need for language-specific filters by polyglot microservices [9,21]. This mesh integration would use Envoy's high-performance Lua filters to perform risk assessment and would interact directly with Istio's Mixer to propagate the policy with lower cross-pod latency and more simplified operations [21,46].

We will test DDoS and chaos-driven attacks with Chaos Mesh and SkyWalking to build resilience to volumetric attacks. We'll be simulating SYN floods, DNS amplification and Redis node partitioning scenarios and will measure DCOTA's capability to withstand policy enforcement and autoscale under load. We will fine-tune the rate-limiting and circuit-breaker thresholds based on observed failure modes, to make sure that both our security and availability objectives are met during sustained high-traffic spikes.

Finally, we want to encode DCOTA's configuration as policy-as-code where we express declarative security rules, risk thresholds, and revocation policies, into GitOps workflows orchestrated by Argo CD. Merge gates with automated compliance scans through OWASP Dependency-Check, Snyk, and static analysis, and alerts about any policy drift due to unintentional modification of critical security settings. This CI/CD automation would also ensure that DCOTA stays auditable, reproducible and compatible with the latest security standards [13,49,50].

Journal of Information Systems Engineering and Management
2025, 10(49s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

References

[1] D. Hardt, “The OAuth 2.0 Authorization Framework,” RFC 6749, IETF, 2012.
[2] OWASP Foundation, “API Security Top 10 2023,” OWASP API Security Project, 2023.
[3] Wikipedia, “Spring Boot,” 2025.
[4] Wikipedia, “JSON Web Token,” May 2015.
[5] Kubernetes v1.22 Release Team, “Kubernetes 1.22: Reaching New Peaks,” Aug. 2021.
[6] Apache JMeter Project, “Apache JMeter User Manual,” 2025.
[7] MaxMind, “GeoLite2 ASN Database,” 2025.
[8] Redis Ltd., “Redis Documentation,” 2025.
[9] PostgreSQL Global Development Group, “PostgreSQL 13 Documentation,” 2025.
[10] Resilience4j Team, “Getting Started with Resilience4j,” 2025.
[11] Mockito Project, “Mockito Framework,” 2025.
[12] JUnit Team, “JUnit 5 User Guide,” 2025.
[13] OWASP Foundation, “OWASP Dependency-Check,” 2025.
[14] HashiCorp, “Vault Documentation,” 2025.
[15] S. Shostack, *Threat Modeling: Designing for Security*, Wiley, 2014.
[16] Wikipedia, “TLS 1.3,” 2025.
[17] RFC 7519, “JSON Web Token (JWT),” IETF, May 2015.
[18] Autho, “Introduction to JSON Web Tokens,” 2025.
[19] OAuth.net, “JWT Usage in OAuth 2.0,” 2025.
[20] OWASP ZAP Project, “Zed Attack Proxy User Guide,” 2022.
[21] PingCap Blog, “Chaos Mesh + SkyWalking,” 2021.
[22] Wikipedia, “Spring Framework,” 2025.
[23] Resilience4j Team, “HikariCP Configuration,” 2025.
[24] IETF, “OAuth 2.0 Token Introspection,” RFC 7662, 2015.
[25] PCI Security Standards Council, “PCI DSS v3.2.1,” 2018.
[26] European Parliament, “Regulation (EU) 2016/679 (GDPR),” 2016.
[27] M. Barnes & P. Jones, “Performance Implications of Token Validation in OAuth APIs,” *IEEE Trans. Serv. Comput.*, 14(2), 2021.
[28] Z. Yi & B. Wu, “Security Enhancement for Microservice APIs with Spring Boot and OAuth,” *Electronics*, 8(9), 2019.
[29] K. Lee & S. Yoo, “Efficient OAuth 2.0 Implementation for High-Volume Financial Services,” *J. Internet Serv. Appl.*, 12(1), 2021.
[30] A. Patel, “Securing Communication Between Spring Boot Microservices Using OAuth2,” *IEEE Cloud Computing*, 5(1), 2022.
[31] Q. Nguyen & O. Baker, “Applying Spring Security and OAuth2 to Protect Microservices,” *J. Software*, 14(6), 2019.
[32] M. de Almeida & E. Canedo, “Authentication and Authorization in Microservices,” *Appl. Sci.*, 12(5), 1703, 2022.
[33] N. Namer, “Analyzing Broken User Authentication Threats to JWT,” Akamai Blog, 2023.
[34] CalState ScholarWorks, “Impact of Performance on JWT Tokens,” 2023.
[35] Akana Blog, “JWT Security Best Practices,” 2018.
[36] Supertokens Blog, “OAuth vs JWT: Key Differences,” 2023.
[37] Wallarm Blog, “Securing Applications with JWT,” 2025.
[38] Sophos, “What Is JSON Web Token Authentication?” 2025.
[39] PortSwigger, “OAuth2 Security Testing Cheat Sheet,” 2024.
[40] E. Frost & D. Holt, “Comparing JWT vs SAML for Fintech APIs,” *J. Internet Services*, 12(4), 2021.
[41] C. Pautasso et al., “RESTful Web Services vs. ‘Big’ Web Services,” *WWW ’08*, 2008.
[42] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.
[43] R. Martin, *Clean Architecture*, Prentice Hall, 2017.

[44] L. Mont & M. Lal, “Data Encryption Techniques for Fintech APIs,” *J. Info Security Appl.*, 45, 2019.
[45] P. Chaudhary & P. Verma, “Token Introspection Performance in OAuth 2.0 Resource Servers,” *Future Gen. Comput. Syst.*, 108, 2020.
[46] R. Ridgway, “API Gateway Patterns for Fintech Microservices,” *IEEE Cloud Computing*, 9(3), 2022.
[47] L. Pons, “Dynamic Rate Limiting for OAuth 2.0 APIs,” *J. Netw. Comput. Appl.*, 204, 103340, 2023.
[48] I. Goldberg, “TLS 1.3: The Next Step in Secure Communications,” *IEEE Commun. Surv. Tutorials*, 22(4), 2020.
[49] J. Bradley, M. Jones & N. Sakimura, “OAuth 2.0 Security Best Practices,” IETF Internet-Draft, 2020.
[50] I. Connolly et al., “JWT Methodologies in Securing Financial Web APIs,” *ACM Comput. Surveys*, 55(6), Art. 115, 2023.