

Performance and Security Analysis of OAuth 2.0 Token Propagation in Spring Boot Microservices

Swathi Gangarapu

Senior Software Engineer & Architect, Virginia, United States

swathigangarapu95@gmail.com

Abstract

This research work evaluates the performance and security implications of OAuth 2.0 token propagation in Spring Boot microservices. Six propagation and validation strategies were tested: local JWT validation, direct token forwarding, gateway relay, per-request introspection, per-request token exchange, and token exchange with a one-second cache. The controlled experiment used Spring Boot 3.5.16 and OpenJDK 17 across four concurrency levels and ten randomized blocks comprising 120,000 measured requests. All 42 predefined behavioural security checks passed and all requests completed successfully. At concurrency 50, local JWT validation achieved a mean batch p95 latency of 8.63 ms and a throughput of 7066.75 requests/s at concurrency 50. Per-request exchange yielded 112.99 ms and 1148.70 requests/s, respectively, whereas cached exchange yielded 23.12 ms and 3677.18 requests/s. Caching reduced exchanged calls from 500 to approximately one per 500 requests batch. Within this single-caller workload, the findings support selective token exchange at authority boundaries and local validation when tokens carry the appropriate audience and scope.

Keywords: OAuth 2.0; Spring Boot ; Micro services; Token Propagation ; JWT ; Spring Security ; Token exchange ; Token introspection ; gateway relay ; Performance evaluation

1. Introduction

In addition to breaking an application into independently deployable components that can be reused across multiple instances, microservice architectures introduce additional authorization boundaries, network hops and opportunities for credential misuse. Securing service to service authentication, authorization and policy enforcement remains a significant challenge in cloud native systems (Minna & Massacci, 2023).

OAuth 2.0 is widely used to protect HTTP APIs through access tokens. Because bearer token generally be used by any party that possess them, their confidentiality, intended audience, lifetime, and secure transport and secure critical considerations (Hardt, 2012; Jones & Hardt, 2012). In distributed systems, JWT access tokens are attractive because resource servers can verify their signatures and claim locally, without contacting the authorization server for every request (Bertocci, 2021). This scalability advantage, however, introduces an additional security consideration i.e a locally validated token may remain usable until its expiration unless mechanisms such as short token lifetimes, sender constrained tokens, revocation controls or online token state validation are employed.

A second issue concerns how an authenticated request and its associated authorization context are propagated across a chain of downstream services. In direct token forwarding, the same credential received by one service is forwarded to subsequent services. Gateway relay centralizes ingress authentication and authorization controls, but it may propagate the same bearer token to downstream services. Token exchange provides an alternative approach in which a service obtain a next downstream specific token with a narrow audience, scope, lifetime and delegation context (Jones et al., 2020). These mechanisms are not mutually exclusive and should not be compared merely as protocol labels. Rather they should be evaluated according to their effects on authorization boundaries, validation operation, network hops, authorization server interactions and credential exposure across the request path.

In this paper, token propagation is therefore examined as a security performance control placement problem. The study does not propose a new OAuth grant type token format. Instead, it presents a standard based comparison supported by an implemented Spring Boot experiment. The evaluation combines threat oriented analysis with repeated measurements of latency, throughput, authorization server call frequency, CPU utilization, memory consumption and behavioural security properties.

1.1 Research Question

RQ1: How do direct token forwarding, gateway relay and token exchange differ with respect to authority confinement, replay exposure and degree of trust placed in downstream services?

RQ2: What are the differences between local JWT validation and online token introspection in terms of processing overhead, revocation freshness and runtime dependency on the authorization server?

RQ3: Under which characteristics and security requirements can a hybrid token propagation strategy reduce authorization server interactions while maintaining bounded downstream authority?

RQ4: What impact do the evaluated strategies have on p95 latency ,p99 latency , throughput , authorization server call frequency and implementation complexity in Spring Boot testbed?

2. Background and Related work

2.1 OAuth 2.0 bearer tokens and JWT access tokens

OAuth 2.0 is a framework for obtaining access tokens for protected resources (Hardt, 2012). The action of using bearer-token is specified in RFC 6750, and it is important to note that merely having the token doesn't require you to disclose it to others, which means that you can make use of it without revealing it to anyone (Jones & Hardt, 2012). To ensure secure deployment, issuer and audience checks, access token lifetime, secure delivery, and limited handling of access tokens are critical. JWT is a JLA that is a small, lightweight format for transmitting claims, such as issuer, subject, audience, expiration, scopes and more, about authorization information. There is an RFC 9068 that specifies a profile for JWT access tokens. Temporal and issuer validation, and signature validation are considered to be local JWT processing, while audience validation is required when there are multiple APIs provided by a single issuer (Bertocci, 2021).

2.2 Local JWT validation, token introspection, and revocation freshness

If there is a local JWT that can be trusted for the signing material, the authorization server is not included in the steady state request path. A resource server checks the signature and application specific and issuer/ temporal/audience claims of the token locally. This is a good way if you can get away with it for high frequency service calls, since you won't have to make an online authorization decision on every call. The main drawback of its operation is its freshness of revocation: an addition to the state check is needed to make sure that a cryptographically valid one doesn't remain valid until the expiry date. The protected resource can request the authorization server if a token is active and get the associated metadata (Richer, 2015) from the authorization server by using the protected resource to request an active token. This online decision may give you a more fresh authorization state than expiry-only local validation, but it includes a network round trip, a dependency on the availability of the authorization-server, and authorization-server processing. In the case of caching introspection results, this results in a reduction of the cost of requests, but also gives an interval of time wherein a revoked credential can be accepted, as a result of the stale state it is in for that interval.

Table 1 distinguishes between the freshness of tokens and the format of tokens. An online JWT mechanism is also a way to evaluate a JWT; for the present experiment, the JWT is evaluated locally and the path to introspection is opaque-token mode is made explicit.

Table 1: Comparison between Local JWT validation and Per-request introspection

Dimension	Local JWT validation	Per-request introspection
Primary request-time work	Local signature and claim validation	Network request plus authorization-server decision
Revocation freshness	Bounded by TTL or extra mechanism	High when authorization-server state is current
Authorization-server request-path dependency	None after signing-key material is available	One online decision per request in this benchmark
Scalability pressure	Primarily resource-server CPU	Shared authorization-server capacity
Failure mode	Can continue while keys remain usable	Requests depend on authorization-server availability

Dimension	Local JWT validation	Per-request introspection
Best fit	High-frequency short-lived audience-bound tokens	Revocation-sensitive decisions

2.3 Token propagation strategies, token exchange and sender constraining

Direct token forwarding is a service that will re-use the caller's access token when calling a downstream service. This means that there's no extra step of communicating with the authorization service, but can increase the replay and/or privilege surface if the same credential is accepted by multiple services or if it has more authority than the downstream service needs. So, it is still crucial to have audience and scope validation. Gateway relay can be used to add an ingress component in front of backend services and to also collect all the routes' authentication, policy enforcement, and observability in one place. But alone, relay does not have any effect on the authority of a bearer token. Services downstream from the gateway should validate the token and only accept a token as proof of permission for the service they are providing, rather than assuming that a token passed through the gateway means that the user is authorized. As described in RFC 8693, token exchange enables a caller to submit a subject token and ask for a new one for a target resource (Jones et al., 2020). The credential exchanged in a micro service call chain can be issued with a more limited audience, scope, lifetime, and delegation context than the original credential, limiting the propagation of extra authority. Any sender-constraining mechanisms, such as DPoP, that mandate the holder of a given token to show proof of an associated key can further reduce the replay value of a stolen token. Table 2 illustrates that neither direct forwarding nor the gateway relay inherently reduces the width of the credential that is presented down the path in steady state. The introduction of control plane work is unique to token exchange to acquire a credential whose scope and audience is aligned with the downstream boundary.

Table 2: Comparison of the three different types of propagation pattern

Criterion	Direct forwarding	Gateway relay	Token exchange
Additional authorization-service call	No	No	Yes at exchange boundary
Credential reused across services	Usually	Usually	No, when exchange produces target token
Authority attenuation	Policy-dependent	Policy-dependent	Directly supported
Ingress centralization	No	Yes	Optional
Authorization-server runtime dependency	Low with local JWT	Low with local JWT	Present at exchange boundary
Primary risk	Broad replay/authority surface	Gateway plus broad relay token	Control-plane latency/availability

2.4 Micro service Security and related work

Run-time security research for cloud microservices emphasizes the continuing importance of service identity, policy enforcement, communication security, and observability (Minna & Massacci, 2023). Empirical work likewise shows that microservice security depends on coordinated practices rather than a single control (Rezaei Nasab et al., 2023). NIST SP 800-207A applies zero-trust principles to cloud-native systems and reinforces explicit, resource-centric authorization decisions rather than implicit trust based on network placement (Chandramouli & Butcher, 2023). Broader work on cloud governance, dependency risk, scalable cloud orchestration, and security-sensitive microservice applications provides contextual evidence that distributed control placement and dependency management matter operationally (Chatterjee, 2021, 2023; Kakaraparthi, 2024; Kubam, 2024). The engineering studies by Das (2021), Madanayake et al. (2015), and Madanishetti et al. (2012) are not OAuth-specific and are retained only as general systems-integration and performance-engineering context rather than as evidence for token-security claims.

3. Methodology

3.1 Research Design and Threat Model

This study has a two part design. A security analysis is first done to a set of standards to map token propagation, validation, choices of the risk of replay exposure, the risk of audience or scope confinement, the risk of revocation freshness, the risk of confused-deputy or over-privilege, the dependency of the availability of the authorization-server, and auditability. Second, Six concrete combinations of strategies are deployed in a controlled experiment of Spring Boot testing end to end request paths. A small application payload is intentionally used for the empirical part to ensure visibility of the effects of Authorisation path and Routing. An HTTP request that is protected across one or more service boundaries is the unit of analysis. The threat model assumes that the traffic will be protected as it is intended during the deployment, but tests the testbed over the loopback HTTP to avoid taking the production network latency for granted. It takes stolen bearer tokens, empty or altered credentials, expired credentials, wrong audiences, wrong scope, too broad an authority in the downstream direction, and the use of credentials with only limited scope in the downstream direction at boundaries to upstream service providers. The values do not imply compromise probabilities nor are they penetration tests.

3.2 Security Evaluation Framework

For the comparative analysis, the security dimensions used throughout have been defined in Table 3. The dimensions used are descriptive (not scored) dimensions. It's important to note that the framework identifies properties that are directly dependent on a mechanism. For example, properties that are specific to a target audience and can be specified using an audience attenuation property in the token exchange, which target audience properties are directly dependent on a token exchange mechanism: from properties that, by contrast, are still dependent on deployment policy. For example, properties that are specified in the JWT validation policy that is revocation-sensitive.

Table 3: Security Dimensions used in comparative Analysis

Dimension	Interpretation	Evidence or control basis
Replay exposure	How broadly a stolen bearer token can be reused.	Token audience, scope, lifetime, and sender constraint.
Authority confinement	Whether downstream authority is narrower than upstream authority.	Audience/scope checks and token-exchange output.
Revocation freshness	How quickly an administrative invalidation can affect acceptance.	Online introspection or short-lived credentials.
Authorization-server dependency	How much the request path depends on authorization-server availability.	Introspection/exchange call frequency.
Delegation traceability	Whether subject/actor or transition context can be recorded.	Exchange semantics and audit design.
Boundary enforcement	Whether unintended services reject a credential.	Behavioral security checks in the testbed.

The key difference between the evaluated propagation strategies is that of the authority which is propagated across service boundaries, not just the strength of the cryptography. Even a well signed token can be unsuitable for a downstream service if the audience or scope is greater than needed. Thus, the proposed approach is analyzed in conjunction with the six other factors: replay exposure, revocation freshness, dependency of the authorization service, delegation traceability, and explicit boundary enforcement. Table 4 shows the explicit threat model aligned to primary controls that are considered in the study. These controls are additive token exchange is in addition to issuer/audience validation sender constraining does not eliminate the need for authorization policy and introspection is in addition to bearer token revocation.

Table 4: Threat Model and Corresponding Control

Threat	Exposure	Primary control	Residual limitation
Bearer-token theft	Replay at any accepting service	Short TTL; narrow audience; DPoP/mTLS	Bearer token remains replayable without sender constraint
Audience confusion	Token accepted by unintended API	Strict audience validation; target-specific exchange	Misconfiguration can defeat intended boundary
Over-scoped token	Unnecessary downstream privilege	Scope attenuation and service authorization	Policy quality depends on authorization-server configuration
Gateway compromise	Misuse of relayed credentials	Independent downstream validation	Gateway remains a high-value component
Authorization-server outage	Introspection/exchange fails	HA, bounded cache, explicit fail policy	Fresh state and exchange depend on control plane
Administrative revocation	Local token remains valid until expiry	Short TTL or online introspection	Freshness-performance trade-off persists

3.3 Performance model and evaluation metrics

Request path is modeled qualitatively as the sum of local security processing, service to service transport serialization, online authorization decisions, token exchange operations, application work and queuing. Let L_{local} denote the cost of processing the local JWT, L_{hop} be the cost of the service to service HTTP, L_{AS} be the cost of processing and transport the JWT from an authorization-service and L_{app} be the minimal cost of the resource-handler. A local resource request is roughly $L_{\text{app}} + L_{\text{local}}$; a direct forwarding requires at least one service hop and one extra validation layer; a gateway relay adds one extra hop and validation layer; per request introspection adds L_{AS} per protected request; per request token exchange adds L_{AS} before the downstream protection, and protection is repeated by the downstream service. When exchanging credentials is cached, the authorization-service contribution may be spread over the number of requests which can use the same downstream contribution. The average exchange contribution is approximately equal to $L_{\text{AS}} / n_{\text{reuse}}$, as long as n_{reuse} tokens are being reused, assuming there are no cache lookup, synchronization or token-lifetime issues. The experiment tests this effect by running per-request exchange and comparing its performance with a cache per subject-token and comparing it to a cache running for a duration of one second in the same topology, upstream/downstream. A variety of complementary measures including latency percentiles, throughput, frequency of calls to the authorization service, CPU Time deltas, memory snapshots and results of behavioural security are not combined in a single score.

3.4 Experimental Procedure and Statistical Analysis

The experiment uses ten randomized blocks with seed 20261002. Within each block, every scenario-concurrency combination is executed in shuffled order. Each measured batch contains 500 timed requests following 100 warm-up requests, with concurrency levels of 1, 10, 50, and 100 in a closed-loop asynchronous Python load generator. Tokens are freshly issued after warm-up for each measured batch, making the hybrid cache cold at the start of measurement. Token issuance, warm-up traffic, resets, metrics retrieval, and security checks are excluded from the timed interval. Client-observed latency is measured from acquisition of the concurrency semaphore until the response body has been read. Throughput is calculated as completed requests divided by measured batch duration. Scenario summaries report the arithmetic mean of the ten batch-level p50, p95, and p99 values at each concurrency. A 95% confidence interval for mean p95 is estimated from 10,000 bootstrap resamples of the ten batch p95 values using seed 20261001. The interval quantifies run-to-run variation on the tested host and is not interpreted as a population-level confidence interval across

machines or deployment environments. The validation package independently confirms 240 measured batches, 120,000 raw request rows, 24 scenario-concurrency summary rows, zero non-200 measured responses, recomputation of raw p95 values, consistency of authorization-service counters, and availability of per-JVM CPU and RSS measurements.

4. Experimental Testbed and Implementation

4.1 Spring Boot architecture and Security Configuration

The implemented testbed uses a shared Spring Boot application JAR started in four roles: authorization service, downstream resource service, upstream service, and gateway relay. Spring Security protects the gateway and upstream services with JWT authentication and protects the downstream resource endpoint with JWT validation and read-scope authorization. The authorization role issues broad or narrow JWTs, maintains opaque-token state for introspection, and exposes a controlled RFC-shaped exchange endpoint. The component is an experimental emulator rather than Spring Authorization Server and therefore must not be presented as a production authorization-server deployment.

Listing 1 shows the role-specific audience-validation logic. The required audience is selected from the service role, so a downstream-only token is rejected when it is presented to the gateway relay or upstream service. This behavior is exercised by the security checks reported in Table 10 .

Listing 1: Role specific issuer and audience validation used by Spring Security Service Resource

```
@Bean
JwtDecoder jwtDecoder() {
    NimbusJwtDecoder decoder = NimbusJwtDecoder.withPublicKey(KeySupport.publicKey()).build();
    OAuth2TokenValidator<Jwt> issuer = JwtValidators.createDefaultWithIssuer("bench-auth");
    OAuth2TokenValidator<Jwt> audience = jwt -> {
        String required = switch (role) {
            case "gateway" -> "gateway";
            case "upstream" -> "service-a";
            case "downstream" -> "service-b";
            default -> null;
        };
        if (required == null || jwt.getAudience().contains(required))
            return OAuth2TokenValidatorResult.success();
        return OAuth2TokenValidatorResult.failure(
            new OAuth2Error("invalid_token", "Missing required audience", null));
    };
    decoder.setJwtValidator(new DelegatingOAuth2TokenValidator<>(issuer, audience));
    return decoder;
}
```

4.2 Token Propagation Implementations

Listing 2 shows the two exchange paths that are central to the performance study. The per-request path requests a new service-b token for every upstream request. The hybrid path uses a concurrent cache keyed by the subject token and reuses the exchanged token for one second. This short TTL is an experimental parameter rather than a recommended universal cache duration.

Listing 2 : Per request and one second cached token exchanged paths in the Upstream Spring Boot Service

```
@GetMapping("/exchange")
public ResponseEntity<String> exchange(@RequestHeader(HttpHeaders.AUTHORIZATION) String
authorization) {
    String token = authorization.replaceFirst("(?i)^Bearer\\s+", "");
    String exchanged = exchangeToken(token);
    return forward(downstreamUrl + "/resource", "Bearer " + exchanged);
}

@GetMapping("/hybrid")
public ResponseEntity<String> hybrid(@RequestHeader(HttpHeaders.AUTHORIZATION) String
authorization) {
    String subject = authorization.replaceFirst("(?i)^Bearer\\s+", "");
    CachedToken c = cached.compute(subject, (key, old) -> {
        long now = System.nanoTime();
        if (old != null && now < old.expiresAtNanos) return old;
        String exchanged = exchangeToken(subject);
        return new CachedToken(exchanged, System.nanoTime() + 1_000_000_000L);
    });
    return forward(downstreamUrl + "/resource", "Bearer " + c.token);
}
```

The exchange endpoint issues a short-lived service-b/read access token only when the subject token is accepted for service-a and contains read permission. Listing 3 shows the relevant guard. The implementation therefore makes scope and audience attenuation observable rather than only conceptual.

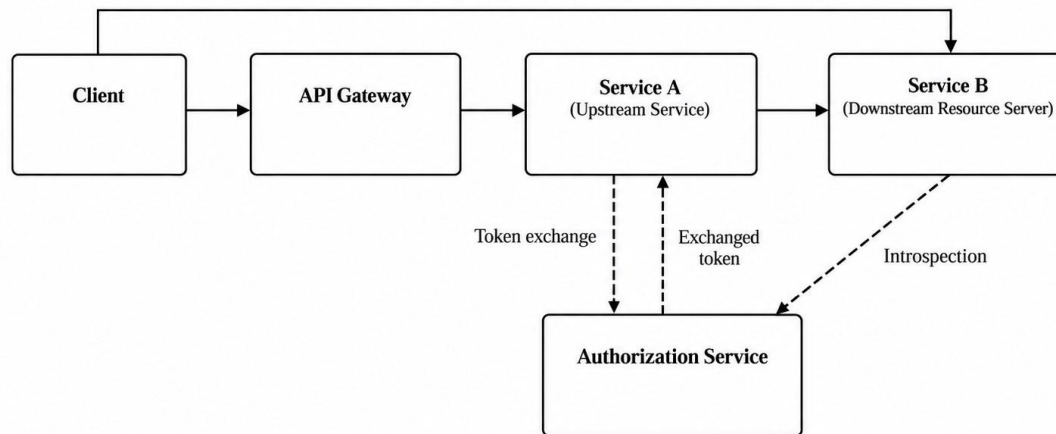
Listing 3: Experimental Exchange guard and narrowed downstream token issuance

```
@PostMapping(value="/exchange", consumes=MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public Map<String,String> exchange(@RequestParam("subject_token") String subjectToken,
    @RequestParam(defaultValue="service-b") String audience) {
    exchangeCalls.incrementAndGet();
    var original = decoder.decode(subjectToken);
    if (!original.getAudience().contains("service-a")
        || !"service-b".equals(audience)
        || !Arrays.asList(original.getClaimAsString("scope").split(" ")).contains("read"))
        throw new ResponseStatusException(HttpStatus.FORBIDDEN, "exchange not permitted");
    return Map.of("access_token", jwt(List.of(audience), "read", 10));
}
```

4.3 Experimental Topology

Figure 1 presents the implemented test bed and separates data-plane request flow from control-plane authorization interactions. The Python load generator acts as the client, while four Spring Boot JVM roles implement the gateway relay, upstream service A, downstream resource service B, and an experimental authorization service. All measured HTTP traffic uses loopback networking. The gateway role is a Spring MVC relay rather than Spring Cloud Gateway, and the authorization role exposes controlled RFC-shaped issuance, introspection, and exchange endpoints rather than a production authorization server. These distinctions are retained to keep the measured system boundary explicit.

Figure 1: Implemented Spring Boot test bed.



Note: Solid arrows represent service-to-service request flow, and dashed arrows represent interactions with the experimental authorization service

Table 5 defines the six measured scenarios. The local JWT path is a single-resource baseline. Direct forwarding adds an upstream hop and preserves the broad caller token. Gateway relay adds a gateway before the upstream service. Introspection uses an opaque token and one authorization-server decision per resource request. Per-request exchange obtains a service-b/read access token for every request. Hybrid cached exchange uses the same exchange topology while allowing bounded reuse of the exchanged credential.

Table 5. Implemented experimental scenarios and request paths

ID	Scenario	Request path	Validation/control	Expected authorization-service calls / 500 requests
S1	Local JWT	Client -> B	Narrow JWT; local validation	0
S2	Direct forwarding	Client -> A -> B	Same broad JWT; local validation at A and B	0
S3	Gateway relay	Client -> Gateway -> A -> B	Same broad JWT; local validation along chain	0
S4	Introspection	Client -> B -> authorization service	Opaque token; online active-state decision	500
S5	Token exchange	Client -> A -> authorization service -> A -> B	Exchange to service-b/read access token on every request	500

ID	Scenario	Request path	Validation/control	Expected authorization-service calls / 500 requests
S6	Hybrid cached token exchange	Same as S5	One-second per-subject exchange cache	Approximately 1

4.4 Environment and Reproducibility

The runtime and experiment parameters are summarized in Table 6. Java 17 is the build target used by the supplied dataset and therefore supersedes the earlier protocol-level Java 21 configuration. Provenance metadata records the runtime versions, host characteristics, randomization seed, workload parameters, and available resource limits.

Table 6. Experimental environment and reproducibility parameters

Parameter	Observed/configured value
Spring Boot	3.5.16
Java runtime/build target	OpenJDK 17.0.20 / Java 17
Maven	3.9.11
Python / aiohttp	3.12.14 / 3.13.5
Host OS	Linux 6.18.44 x86_64, glibc 2.39
CPU	AMD EPYC 9V74; reported 9 CPUs; cgroup quota 800000/100000
Memory	Host report 10.45 GB; cgroup memory maximum 8 GB
Runs	10 randomized blocks
Requests per measured batch	500 after 100 warm-up requests
Concurrency	1, 10, 50, 100
Randomization seed	20261002
Bootstrap seed / resamples	20261001 / 10,000

5. Results

5.1 Performance results

Table 7: Mean batch p95 latency (ms) across client concurrency levels; n = 10 batches per cell

Scenario	C=1	C=10	C=50	C=100
Local JWT	1.05	3.97	8.63	38.11
Direct forwarding	3.07	6.40	43.19	37.38
Gateway relay	7.10	14.06	42.53	60.04
Introspection	4.62	8.09	42.77	91.47
Token exchange	8.09	18.88	112.99	156.54
Hybrid cached exchange	2.62	7.76	23.12	48.45

Table 7 shows that local JWT validation had the lowest mean p95 at concurrency 1, 10, and 50. At concurrency 100, direct forwarding (37.38 ms) and local JWT (38.11 ms) were close, but the local-JWT confidence interval was wide (14.70-72.15 ms), indicating substantial run-to-run variability. Per-request token exchange had the highest mean p95 at every concurrency level and reached 156.54 ms at concurrency 100. Figure 2 presents the same relationship as a scaling curve.

Figure 5: Mean batch p95 latency for the six scenarios at client concurrency 1, 10, 50, and 100

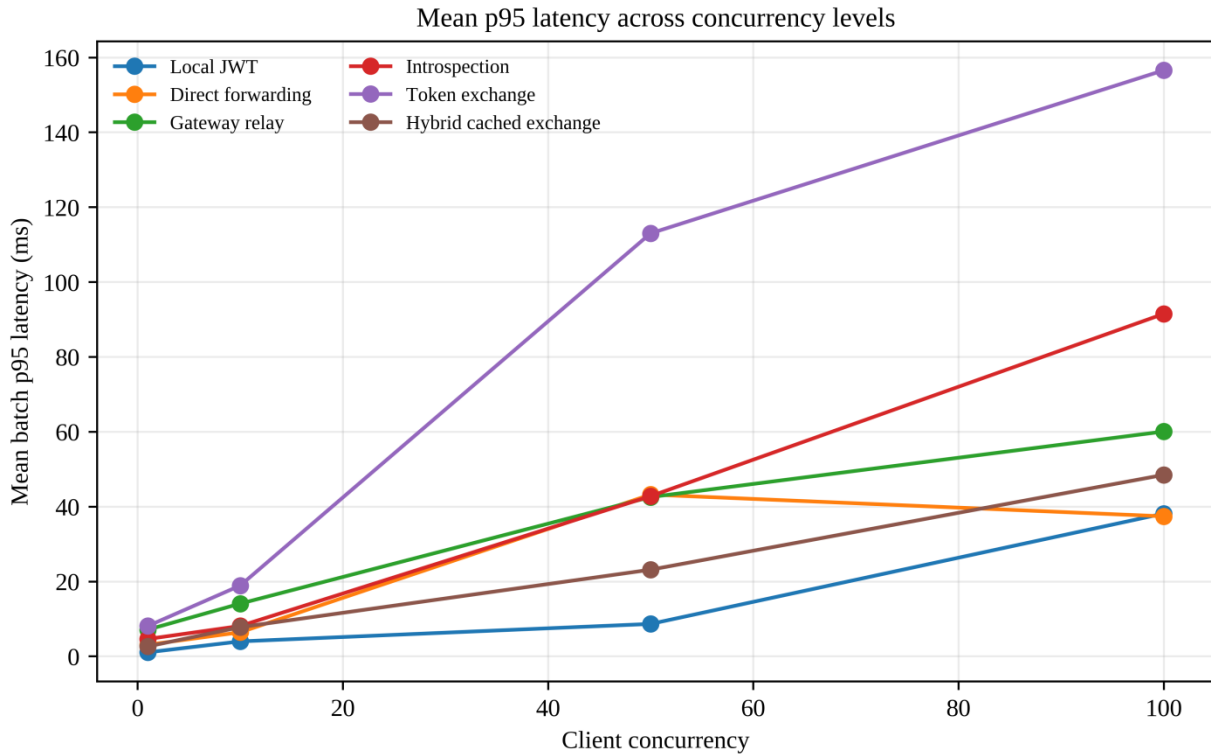
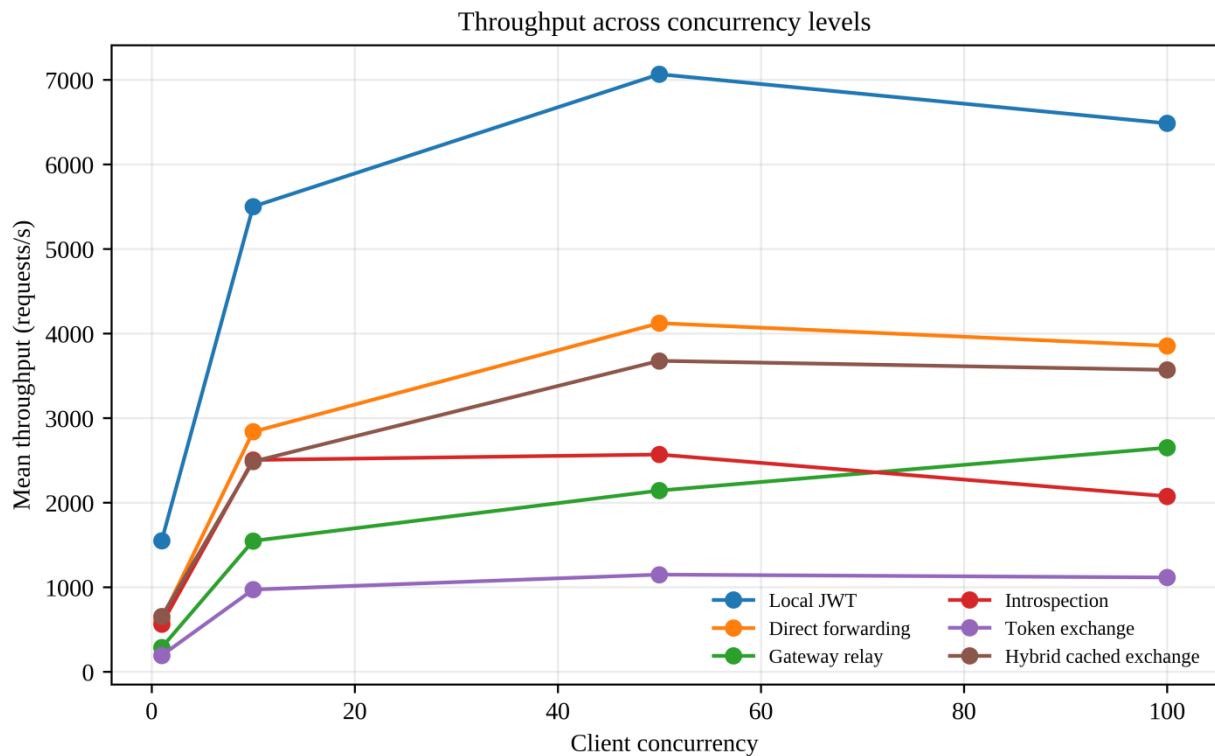


Table 8. Mean throughput (requests/s) across client concurrency levels; n = 10 batches per cell

Scenario	C=1	C=10	C=50	C=100
Local JWT	1549.36	5501.42	7066.75	6485.61
Direct forwarding	618.65	2838.15	4122.52	3854.62
Gateway relay	285.86	1547.43	2142.46	2649.79
Introspection	562.04	2504.58	2568.91	2075.00
Token exchange	192.47	971.46	1148.70	1114.85
Hybrid cached exchange	652.76	2485.86	3677.18	3569.82

Throughput results in Table 8 and Figure 3 show the complementary cost pattern. Local JWT validation produced the highest mean throughput at all four concurrency levels, reaching 7066.75 requests/s at concurrency 50. Per-request exchange remained the lowest-throughput path, reaching 1148.70 requests/s at concurrency 50 and 1114.85 requests/s at concurrency 100. Hybrid cached exchange substantially outperformed per-request exchange because repeated authorization-server exchanges were avoided after the first eligible request in a batch.

Figure 3: Mean throughput across client concurrency levels



Note: Values are means of ten finite 500-request batches and are not steady-state saturation estimates

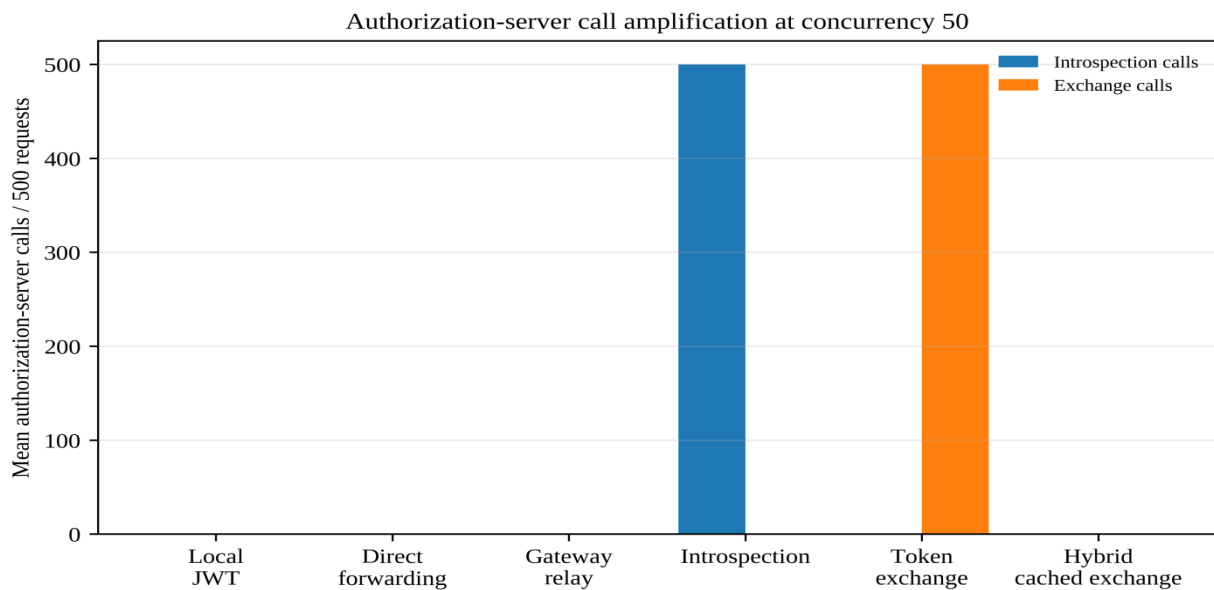
Table 9. Detailed performance and authorization-service results at client concurrency 50

Scenario	Mean p50 latency (ms)	Mean p95 latency (ms)	Mean p99 latency (ms)	Mean throughput (requests/s)	Mean authorization-service calls / 500 requests
Local JWT	4.88	8.63	10.81	7066.75	Introspection 0; exchange 0.0
Direct forwarding	10.70	43.19	48.84	4122.52	Introspection 0; exchange 0.0
Gateway relay	21.96	42.53	50.66	2142.46	Introspection 0; exchange 0.0
Introspection	18.31	42.77	57.29	2568.91	Introspection 500; exchange 0.0
Token exchange	48.38	112.99	153.84	1148.70	Introspection 0; exchange 500.0
Hybrid cached token exchange	11.03	23.12	28.73	3677.18	Introspection 0; exchange 1.0

At concurrency 50, Table 9 shows a particularly clear exchange comparison. Hybrid cached exchange reduced mean p95 latency from 112.99 ms to 23.12 ms, a 79.5% reduction relative to per-request exchange, while mean throughput increased from 1148.70 to 3677.18 requests/s (220.1% higher). Mean exchange calls fell from 500 to 1.0 per 500

measured requests, corresponding to a 99.8% reduction in exchange-call frequency under this single-caller, short-batch workload. Figure 4 visualizes the authorization-server interaction pattern across all scenarios.

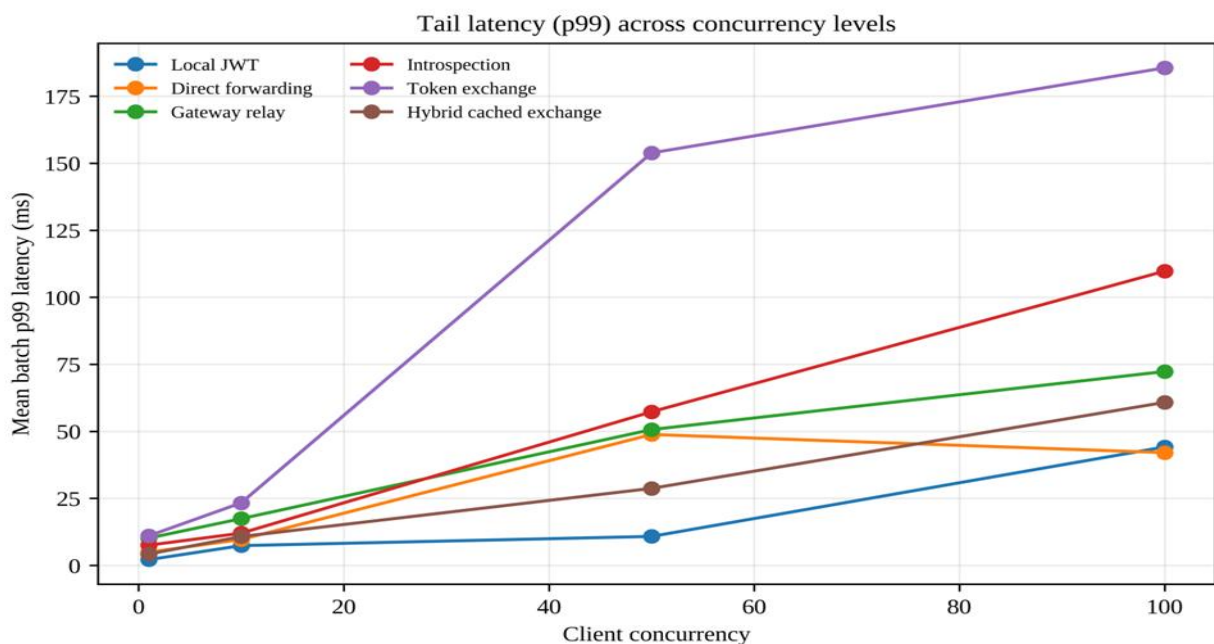
Figure 4: Mean introspection and token-exchange calls per 500 measured requests at concurrency 50



Note: The hybrid result is specific to one subject token per measured batch and a one-second cache TTL

Tail latency is also material. Figure 5 shows mean batch p99 across concurrency levels. At concurrency 100, token exchange reached 185.56 ms p99, introspection reached 109.74 ms, gateway relay reached 72.31 ms, hybrid cached exchange reached 60.80 ms, local JWT reached 44.17 ms, and direct forwarding reached 42.06 ms. The finite-batch design and shared-host execution mean that these values describe the tested paths and host conditions rather than production service-level objectives.

Figure 5: Mean batch p99 latency across concurrency levels, showing tail-latency growth in authorization-service-dependent paths at higher client concurrency



5.2 Behavioral security validation

Table 10: Behavioral security-validation summary

Scenario	Behavioral checks executed	Checks passed
Local JWT	7	7/7
Direct forwarding	7	7/7
Gateway relay	7	7/7
Introspection	3	3/3
Token exchange	11	11/11
Hybrid cached exchange	7	7/7

All 42 predefined behavioral checks passed, as summarized in Table 10. The checks verify expected handling of valid, missing, malformed, tampered, expired, wrong-audience, and no-read-scope credentials where applicable. The exchange-specific tests additionally verified that the issued token was reduced to service-b/read, that this downstream-only token was rejected at the upstream and gateway services, and that an unauthorized exchange target was denied. These test provides concrete evidence that the benchmark enforces the intended audience and scope transitions, but they are not penetration test or a complete OAuth conformance suit.

6. Discussions

6.1. RQ1: Strategy for propagation and limitation of authority

The broad caller token was maintained in the implemented testbed by using direct forwarding and gateway relay. Their primary security property is thus that they are easy to operate, rather than that they have fewer and fewer rights. The security boundary was changed with token exchange since it issued a service-b/read access token, and it denied access to that token at service-a and the gateway. Checks on security ensure desired fringe actions. The analysis thus confirms that token exchange can be useful as a tool for controlling the spreading of downstream authority under any authorization policy that restricts audience and scope, but does not imply that exchange is an automatic secure policy.

6.2 RQ2: Validation strategy, revocation freshness, and overhead

At steady state measured path, local JWT validation was observed to avoid sending a call to the authorization-service and yielded the largest throughput with the various concurrency levels tested. The authorization server was included in every protected request, resulting in 500 introspection requests per 500 measured requests: per-request introspection made the authorization server a part of each of the protected requests, causing 500 introspection requests for every 500 requests measured. This design provides a more clean and centralized decision making but also also opens up the request delay and availability to the authorization-server behavior. The experiment did not run an event of revocation nor did it simulate an outage of the authorization-server so the measure of revocation responsiveness/behavior on failure is still an architectural expectation.

6.3 RQ3: Conditions favoring hybrid model

The format that worked best was the hybrid (caller token that produced follow-up eligible calls to one target audience down). When in that state, the second case (one-second caching) was able to decrease exchange traffic by 99.8% at concurrency 50 and resulted in significant lower latency than exchange per request. This discovery encourages the idea of exchanging over valid reuse, and the experiment tested one subject token per batch measured. Different balances of cache hit ratio and security-performance may result in different deployments with many callers, multiple audiences, policy changes that are rapidly updated or policy changes that are rapidly revoked, or different types of actions that are revoked

6.4 RQ4: Measurable effects in the system in the Spring Boot implementation

These differences are not only in the cryptographic validation of the requests, but also in the full path implemented by the request. There are differences in the number of HTTP hops, serialization, Spring Security processing and control plane interaction in the different scenarios, including direct forwarding, gateway relay, introspection and exchange. The results quantify the behavior of end-to-end paths in this testbed in general, rather than just one of the token-processing primitives.

6.5 Security-performance synthesis and hybrid design implications

The empirical results and threat-oriented analysis indicate a policy-constrained hybrid propagation model instead of single propagation model. Short-lived JWTs can be used for local validation, if the authority presented is suitable for the target service, and they are targeted to a specific audience of users. Short-lived JWTs can be used for local validation, when the authority presented is appropriate for the target service, and the audience of users is defined. When a call crosses a privilege/ trust/ sensitivity boundary, it can get a downstream specific credential by using the token exchange. Online introspection is most relevant where the state is more important than to avoid a control-plane round trip. It should be subject to constraints of audience, scope, subject, lifetime and cache-isolation when it is reused.

Table 11: Decision conditions for the hybrid propagation model

Operation class	Propagation	Validation	Additional control
High-volume internal read	Audience-bound JWT	Local JWT	Strict issuer/audience; short TTL
Sensitive privilege change	Token exchange	Local JWT; optional fresh state check	Narrow scope; short TTL; strong audit
Cross-domain call	Token exchange	Per trust agreement	Explicit target audience; sender constraint
Revocation-critical endpoint	Existing or exchanged token	Introspection	Authorization-server high availability; explicit fail policy
User request fan-out	Gateway plus selective exchange	Local JWT downstream	Do not propagate refresh token

Table 11 does not rank mechanisms but rather represents conditional design choices. The hybrid model used assumed repeated requests from a single caller token to a single downstream audience using a one second cache. For multi-tenant and/or multi-audience caches, each key must be specifically analyzed for its isolation, eviction and revocation, token lifetime and caller diversity, so the measured benefit should not be extrapolated to other cases. None of the assessed strategies scores higher than all the objectives. Instead, the design is to include online validation of states or privilege transitions where the security gain is more valuable than the added dependency on the control-plane, and to include locally validatable and narrowly scoped credentials where the call can be made without an online call to the target service to make an adequate authorization decision. This is in line with the measured cost of per-request interactions with authorization service, and the observed benefit of bounded exchange reuse under the workload tested.

7. Limitations

There are a number of limitations to this study. First, the experiments were performed on a single host with loopback networking and no TLS/mTLS, container networking, databases, service meshes or realistic inter-host latency. Thus, the timings reported for latency and throughput should not be considered production estimates. Secondly, there are different service topologies in the six scenarios. The difference in performance could then be due to any other network hops, routing, serialisation or token propagation mechanisms, plus any additional validation that may have been needed. If you wanted to isolate any overheads related to tokens, then you would need to use a matched-topology design. Third, the authorization server and gateway were not intended to be production grade, but were streamlined experiments. Security testing covered only the following features: token validity, claims, scopes, exchange attenuation, and token rejection

cases; other features, including key rotation, replay protection, and outages were not tested. Finally, results were based on ten finite batches on one machine. Therefore, the values for confidence intervals, CPU and memory measurements should not be extrapolated to other hardware, cloud platforms, JVMs or production workloads.

8. Conclusion

OAuth 2.0 token propagation between spring Boot microservices may be viewed as a control placement problem in which performance, authority containment, revocation freshness and control plane dependency are interrelated. The results support the use of token exchange when authority boundaries exchange, but not at every method call, and support local JWT validation when the token is already appropriate for the target resource. When additional centralized token state verification is required, introspection remains a suitable option. However, these conclusions are specific to the experimental testbed and threat model. Production deployment decisions should additionally account for TLS, sender constraining, authorization server resilience, cache isolation, revocation policy, service identity, key rotation and workload diversity.

9. Declaration

Data Availability: The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Funding: This research received no specific grant from any funding institute.

Ethical approval: No human participants were involved in this study therefore, no ethical approval is required.

Conflict of Interest: The author declares no conflict of interest regarding the publication of this research work.

References:

1. Bertocci, V. (2021). JSON Web Token (JWT) profile for OAuth 2.0 access tokens (RFC 9068). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9068>
2. Chandramouli, R., & Butcher, Z. (2023). A zero trust architecture model for access control in cloud-native applications in multi-cloud environments (NIST SP 800-207A). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-207A>
3. Chatterjee, S. (2021). Mitigating supply chain malware risks in operational technology: Challenges and solutions for the oil and gas industry. *Journal of Advances in Developmental Research*. <https://doi.org/10.5281/zenodo.14551828>
4. Chatterjee, S. (2023). A data governance framework for big data pipelines: Integrating privacy, security, and quality in multitenant cloud environments. *Technix International Journal for Engineering Research*, 10(5), 746-758. <https://doi.org/10.56975/tijer.v10i5.158181>
5. Das, P. (2021). Optimizing sensor integration for enhanced localization in underwater ROVs. *International Journal of Scientific Research in Engineering and Management*. <https://doi.org/10.55041/IJSREM10901>
6. Fett, D., Küsters, R., & Schmitz, G. (2016). A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1204-1215). Association for Computing Machinery. <https://doi.org/10.1145/2976749.2978385>
7. Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., & Waite, D. (2023). OAuth 2.0 Demonstrating Proof of Possession (DPoP) (RFC 9449). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9449>
8. Hardt, D. (Ed.). (2012). The OAuth 2.0 authorization framework (RFC 6749). Internet Engineering Task Force. <https://doi.org/10.17487/RFC6749>
9. Jones, M., & Hardt, D. (2012). The OAuth 2.0 authorization framework: Bearer token usage (RFC 6750). Internet Engineering Task Force. <https://doi.org/10.17487/RFC6750>
10. Jones, M., Nadalin, A., Campbell, B., Bradley, J., & Mortimore, C. (2020). OAuth 2.0 token exchange (RFC 8693). Internet Engineering Task Force. <https://doi.org/10.17487/RFC8693>
11. Kakaraparthi, G. C. (2024). Integrating serverless architectures and Kubernetes for scalable and high-availability AI workflows. *International Journal of Intelligent Systems and Applications in Engineering*, 12(4), 5896-5905. <https://doi.org/10.6084/m9.figshare.30445046>

12. Kubam, C. S. (2024). Agentic AI microservice framework for deepfake and document fraud detection in KYC pipelines. *Journal of Information Systems Engineering and Management*, 9(1). <https://doi.org/10.5281/zenodo.18009551>
13. Madanayake, A., Cintra, R. J., Dimitrov, V., Bayer, F., Wahid, K. A., Kulasekera, S., Edirisuriya, A., Potluri, U., Madishetty, S., & Rajapaksha, N. (2015). Low-power VLSI architectures for DCT/DWT: Precision vs. approximation for HD video, biomedical, and smart antenna applications. *IEEE Circuits and Systems Magazine*, 15(1), 25-47. <https://doi.org/10.1109/MCAS.2014.2385553>
14. Madanishetti, S., Madanayake, A., Cintra, R. J., Mugler, D., & Dimitrov, V. S. (2012). Error-free VLSI architecture for the 2-D Daubechies 4-tap filter using algebraic integers. In *2012 IEEE International Symposium on Circuits and Systems* (pp. 1484-1487). IEEE. <https://doi.org/10.1109/ISCAS.2012.6271528>
15. Minna, F., & Massacci, F. (2023). SoK: Run-time security for cloud microservices. Are we there yet? *Computers & Security*, 127, 103119. <https://doi.org/10.1016/j.cose.2023.103119>
16. Rezaei Nasab, A., Shahin, M., Hoseyni Raviz, S. A., Liang, P., Mashmool, A., & Lenarduzzi, V. (2023). An empirical study of security practices for microservices systems. *Journal of Systems and Software*, 198, 111563. <https://doi.org/10.1016/j.jss.2022.111563>
17. Richer, J. (Ed.). (2015). OAuth 2.0 token introspection (RFC 7662). Internet Engineering Task Force. <https://doi.org/10.17487/RFC7662>
18. Venčkauskas, A., Kukta, D., Grigaliūnas, Š., & Brūzgienė, R. (2023). Enhancing microservices security with token-based access control method. *Sensors*, 23(6), 3363. <https://doi.org/10.3390/s23063363>