The American Journal of Engineering and Technology
ISSN 2689-0984 Volume 08 - 2026

Designing Secure Microservices on Kubernetes: An Architectural Deep Dive into OAuth2, PKCE, Keycloak, Vault, and LDAP

Igor Zuykov
Chief Software Engineer, G-71 Inc Sugar Hill, GA, USA

Received: 06 Feb 2026 | Received Revised Version: 13 Mar 2026 | Accepted: 25 Apr 2026 | Published: 12 May 2026

Volume 08 Issue 05 2026 | Crossref DOI: 10.37547/tajmei/Volume08Issue05-07

Abstract

The article examines an architectural model for securing microservice systems in a Kubernetes environment, integrating mechanisms for federated authentication, secrets management, network isolation, and secure data evolution, leveraging OAuth 2.1, PKCE, Keycloak, Vault, and LDAP. This is motivated by the increasing attack surface of cloud-native environments, exacerbated by container ephemerality, flat cluster networking, and fragmented access domains. In such environments, a compromised container can attack other services, steal credentials, and intercept traffic. The paper thus seeks to develop and validate an integrated architectural approach to secure the Kubernetes microservice ecosystem under the Zero Trust security model. The main contribution of the article is formalizing an end-to-end security pattern for the IAM domain. This pattern is formed by combining several technologies (PKCE, mTLS, Vault PKI, External Secrets Operator, and Liquibase) into a single model. It is concluded that the combination of an API Gateway, Keycloak with LDAP, short-lived secrets, and separated privileges for applications and migrations reduces the risk of token interception, secret leakage, and lateral movement within the cluster at an acceptable infrastructure cost. The article will be useful for IT architects, DevSecOps engineers, information security specialists, and cloud platform developers.

Keywords: microservice security, Kubernetes, OAuth 2.1, PKCE, Keycloak, Vault, LDAP, Zero Trust, mTLS.

Cite This Article: Zuykov, I. (2026). Designing Secure Microservices on Kubernetes: An Architectural Deep Dive into OAuth2, PKCE, Keycloak, Vault, and LDAP. The American Journal of Engineering and Technology, 8(05). https://doi.org/10.37547/tajmei/Volume08Issue05-07

Introduction

The adoption of cloud computing technologies, as well as the transition from monolithic to microservice-based architectures, has changed the way enterprise software applications are developed, deployed, and managed (Faustino et al., 2024). With the Kubernetes platform being the primary solution for container orchestration, developers can easily scale workloads up and down, recover from failures, and enable high fault tolerance by distributing workloads across multiple availability zones (Nascimento et al., 2024). Unfortunately, the same design decisions that have made Kubernetes such a powerful and flexible tool have also opened new, highly dangerous attack surfaces (Chen et al., 2025). The flat networking model of Kubernetes, in which interaction between any container and any other cluster node is permitted by default, erases traditional perimeter trust boundaries in full (Cesarano & Natella, 2025). The ephemerality of container workloads leads to constant rotation of IP addresses and node identifiers, rendering

classical firewalls and static IP allowlists ineffective and inapplicable under current conditions (Hussain et al., 2025).

The central problem of contemporary distributed infrastructures is the decentralization of access control and security configuration management (Xi et al., 2023). Traditional yet outdated security antipatterns, such as independent validation of JSON Web Tokens within each microservice, storing database credentials in static environment variables, or using standard Kubernetes Secrets objects, are often the root cause of large-scale system compromises. In addition, the use of outdated authorization flows, such as the Implicit Flow for single-page web applications, makes the architecture vulnerable to token interception, cross-site request forgery, and cross-site scripting attacks (Rahat et al., 2022). Under increasing regulatory pressure, such architectural deficiencies create insurmountable barriers to the successful completion of information security and compliance audits.

This study addresses the need to shift from reactive protection methods to a proactive paradigm embedded directly into the architectural foundation. The problem lies in the lack of unified, empirically validated reference models that integrate identity management, network traffic, secrets, and databases into a single, coordinated domain (Saleh et al., 2025).

The purpose of this study is to develop, substantiate, and analyze a comprehensive architectural model for the end-to-end protection of microservice ecosystems in Kubernetes.

To achieve this goal, the following objectives were formulated:

1. To synthesize the concept of continuous verification with modern identity federation protocols based on Keycloak and enterprise LDAP directories.
2. To assess the protection mechanisms for public clients using the current OAuth 2.1 standard and the cryptographic Proof Key for Code Exchange extension.
3. To analyze the performance and resource intensity of traffic protection patterns by comparing centralized gateways and decentralized service meshes.
4. To design a secure, automated domain for lifecycle management of secrets and database schema evolution using HashiCorp Vault and Liquibase.

The scientific novelty of the work lies in the articulation and formalization of an end-to-end security pattern that removes the fragmentation of integration solutions. For the first time, a single analytical context brings together empirical latency indicators for PKCE implementation, Demonstrating Proof of Possession (DPoP) mechanisms, and a structured methodology for zero-downtime database schema evolution based on backward-compatible migration patterns.

2. Materials and Methodology

The study is based on the analytical processing of a body of sources covering research on cloud-native architectures, Kubernetes security, federated authentication, secrets management, inter-service traffic protection, and DevOps practices for data schema evolution. The material base included works devoted to migration from monoliths to microservices and the operational characteristics of cloud-native systems (Faustino et al., 2024. Nascimento et al., 2024), studies of the attack surface and behavioral monitoring in Kubernetes (Cesarano & Natella, 2025. Chen et al., 2025), publications on decentralized access control and identity-centric security in CI/CD pipelines (Xi et al., 2023. Saleh et al., 2025), as well as works on modern OAuth 2.1 flows, PKCE, and user session protection (Hardt et al., 2024. Rahat et al., 2022. Singh & Chaudhary, 2023. Chmelev, 2025). A separate block of materials was formed by studies on data and privilege management in microservices, integration of external resources with Kubernetes, service meshes, Kafka protection, database migrations, and canary deployments (Laigner et al., 2021; Lublinsky et al., 2022; Song et al., 2024; Meka, 2025; Malhotra et al., 2024; Ebad & Amara, 2026).

The research methodology combines architectural synthesis, comparative analysis, and interpretation of empirical metrics drawn from domain-specific works. These layers of protection were compared in the literature survey in the context of a Kubernetes microservice architecture. This included Keycloak with LDAP federation for IAM using OAuth 2.1 + PKCE, mTLS for east-west traffic, centralized secrets management with Vault, and Kubernetes operator patterns to deliver these capabilities (Hardt et al., 2024; Hussain et al., 2025; Lublinsky et al., 2022). Architectural antipatterns such as storing secrets as plain Kubernetes Secret objects, validating tokens locally in microservices, and giving excessive database access

permissions were identified and weighed against alternative solutions such as Zero Trust, the Principle of Least Privilege, and a resilient CI/CD domain supplemented by Liquibase and Istio (Laigner et al., 2021; Ebad & Amara, 2026; Malhotra et al., 2024). This approach enabled integrating identity protocols, network mechanisms, secrets, and the data lifecycle into a unified research model suitable for the architectural assessment of protected microservices.

3. Results and Discussion

The concept of Zero Trust Architecture postulates the complete rejection of implicit trust based on a component's network location. In the context of Kubernetes clusters, this means that network interaction between internal microservices, for example, Service A and Service B, cannot be regarded a priori as legitimate and secure merely because both containers operate within a single protected corporate network. Figure 1 presents the high-level architecture of a secured microservices model, implementing a strict multilayered security and routing approach.

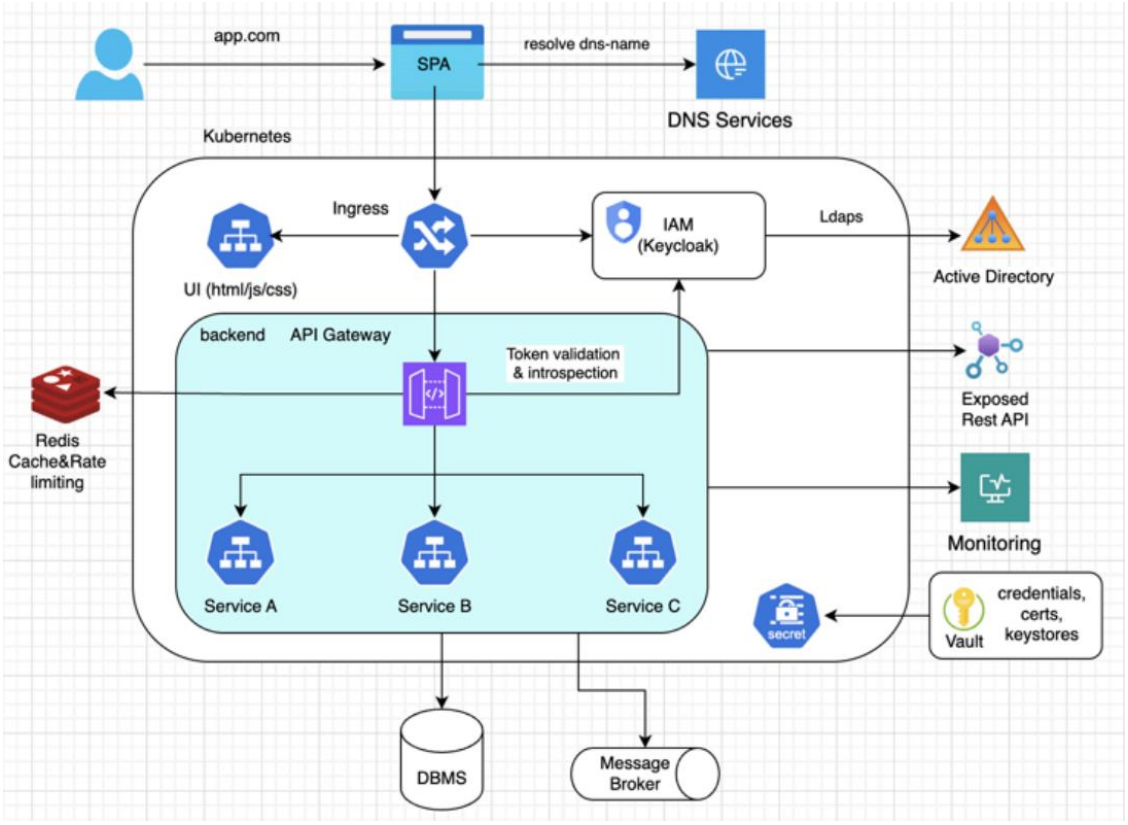

Fig. 1. Secured MSA High-Level Architecture

The system architecture shown in Figure 1 is a representation of a domain infrastructure split into functional zones and trust zones. The interaction is initiated on the client side, when a user accesses the domain app.com, which hosts a single-page application (SPA). After domain name resolution is handled by DNS Services, the entry point for all incoming traffic to Kubernetes is the Ingress component of the cluster. Ingress handles termination of incoming external HTTP(S) requests and is the primary routing option. Static resources for the frontend, such as the user interface (HTML, JavaScript and CSS) are deployed alongside the backend services on the Kubernetes infrastructure.

The central traffic control node is the Backend/API Gateway zone, highlighted in blue. The incoming flow from Ingress enters the API Gateway, which functions as a single intelligent access control point for the underlying microservices, Service A, B, and C. To optimize performance and protect against DDoS attacks, the API

Gateway is integrated with Redis, an in-memory store. In this configuration, Redis performs a dual function. It caches commonly requested data and rate-limits requests to prevent internal APIs from being overwhelmed.

The security foundation is implemented in the Identity and Access Management block as the Keycloak server, which is closely coupled with Ingress and API Gateway to enable constant token verification and introspection. Keycloak integrates with an existing Active Directory using LDAPS, enabling corporate single sign-on (SSO). Other infrastructure components include a database, message broker, central observability and monitoring solution, and HashiCorp Vault. Vault is a centralized secret store that allows for credentials, certificates, and key stores to be easily injected into Kubernetes pods.

Transport-layer network security at L3/L4 requires a Container Network Interface (CNI) plugin that enforces network policies. Some CNI plugins (e.g., Calico) rely on iptables rule chains, which scale poorly under high concurrency. More modern CNI implementations, such as Cilium, use kernel-embedded filtering and maintain high throughput even with many policies.

Yet empirical data also indicate that attempting to implement deep packet inspection at the application layer (L7) within the CNI itself results in a catastrophic drop in throughput. This provides the architectural rationale for moving L7 security tasks, token validation, and Rate Limiting to a specialized API Gateway, while leaving the CNI with the exclusive task of microsegmentation at L3/L4.

The second critical line of defense for the microservice ecosystem is user session management. Historically, single-page applications widely employed the simplified Implicit Flow grant (Singh & Chaudhary, 2023). Within this flow, the access token was returned to the client directly in the URI fragment, making it vulnerable to interception via browser history, HTTP referrers, and malicious scripts on the page.

The contemporary cyberthreat landscape has dictated a revision of standards. The current specifications of the OAuth 2.1 standard prohibit the use of Implicit Flow, as well as the Resource Owner Password Credentials flow, in which the client application receives the user's login and password directly (Hardt et al., 2024). In the architecture of contemporary enterprise software, the de facto standard for public clients, which are incapable of securely storing a static client_secret, becomes the Authorization Code Flow with mandatory use of the PKCE cryptographic extension. The detailed mechanics and role distribution logic of this process are visualized in Figure 2.

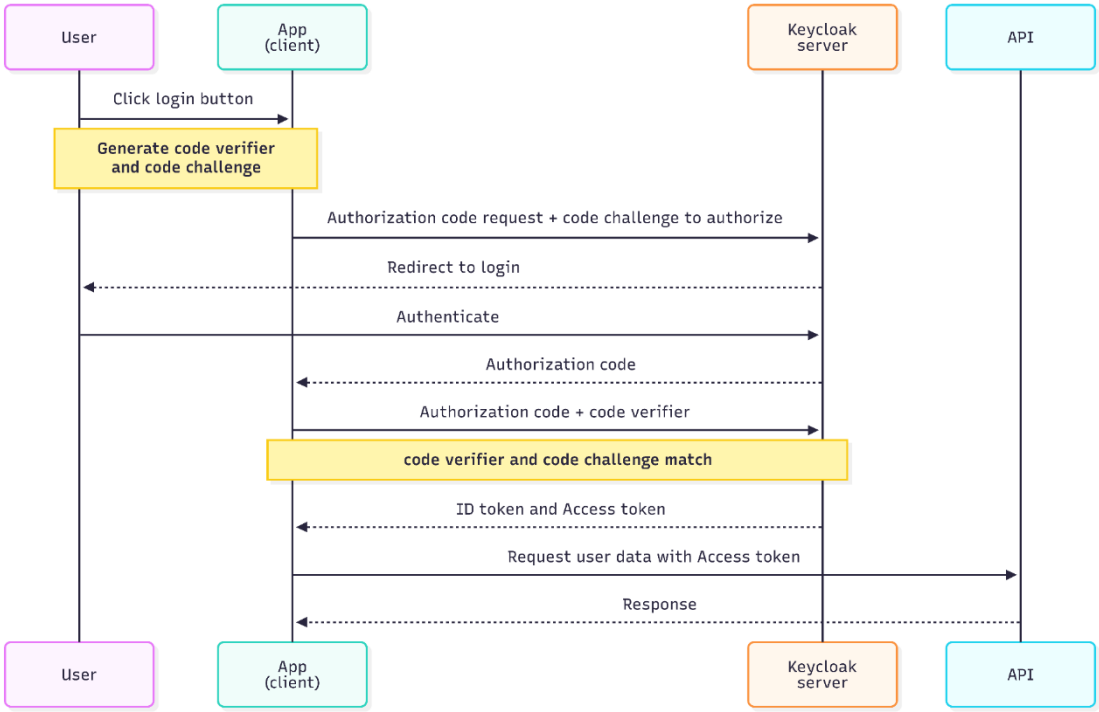

Fig. 2. Authorization Code Flow with PKCE

Figure 2 clearly reveals the multistage mechanism of protection against code interception. At the beginning of the process, the user initiates sign-in through the application interface. The key feature of PKCE manifests itself as early as the second step. Part of the security logic is shifted to the client side. The application generates a cryptographically strong random string, code_verifier, and computes a derived value, code_challenge, from it, usually using the SHA-256 hashing algorithm. The client then sends an authorization request to the Keycloak server, which must include the computed code_challenge. Keycloak redirects the user to the login form, where authentication occurs directly on the identity server, for example, via integration with corporate Active Directory.

The client application never comes into contact with the user's password. After successful authentication, Keycloak returns the intermediate artifact authorization_code. This code by itself does not grant access to data. To obtain tokens, the client sends a second request to Keycloak, passing the received code and, critically, the original code_verifier. On the server side, a critical cryptographic check is performed. Keycloak then hashes the code_verifier and compares the result to the code_challenge it sent earlier. It follows from the way the proof was constructed that this proves that the token exchange request came from the same client that started the process. This mechanism completely neutralizes Authorization Code Interception attacks. If successful, the server returns the ID token with the user profile as well as an Access token that enables access to the protected API.

Keycloak can integrate with Microsoft Active Directory (AD) over the secure LDAPS transport protocol. When an user is authenticated, AD group memberships are retrieved and mapped to application roles stored as an array in the JSON web token (JWT). Role mapping allows role-based access control. Keycloak Initialization in React is shown in Figure 3.

```
const initKeycloak = (onAuthenticatedCallback: any): void => {
    keycloak = new Keycloak({
        url: 'https://keycloak.example.com/auth',
        realm: 'myrealm',
        clientId: 'spa-client'
    });

    keycloak.init({
        onLoad: "login-required", // 'check-sso'
        pkceMethod: "S256",  // enables PKCE
    })
        .then((authenticated) => {
            if (!authenticated) {
                console.error("user is not authenticated..!");
            }
            onAuthenticatedCallback();
        })
        .catch(console.error);
};
```

Fig. 3. Keycloak Initialization in React

Recommendations to improve session security include Refresh Token Rotation and Proof-of-Possession of refresh tokens. RTR guarantees that each session refreshes issues a new refresh token while invalidating the previous one, protecting against replay attacks. DPoP cryptographically binds the token to a client's specific key pair. The client proves possession of the private key for each request, preventing token replay even if the

token itself is intercepted. Analysis of empirical data indicates that applying these methods has minimal impact on user experience (Chmelev, 2025). Impact of modern OAuth 2.1 cryptographic extensions on authorization latency is shown in Table 1.

Table 1. Impact of modern OAuth 2.1 cryptographic extensions on authorization latency

Security Mechanism	Architectural Impact Description	Average Overhead, Latency	Estimated UX Impact
PKCE, SHA-256	Generation of the code_verifier, client-side hash computation, and server-side verification	~16 ms	Negligible
DPoP, Proof-of-Possession	Generation and verification of a DPoP JWT signature for each HTTP request	~9 ms	Not noticeable to the user
Total Overhead	Comprehensive session protection against interception and token cloning	~25 ms	The delay is fully offset by the security gains

As shown in Table 1, the total delay of 25 milliseconds is a small and acceptable cost for the fundamental elimination of session token theft risks. For additional protection, tokens should be stored in sessionStorage or managed using the Backend-for-Frontend architectural pattern with cookies set to the HttpOnly flag, which fully prevents theft via XSS vulnerabilities in localStorage.

In a distributed Kubernetes system, network trust is divided into two independent vectors: North-South traffic, which is external and inbound or outbound, and East-West traffic, which is internal inter-service communication. API Gateway and mutual TLS authentication based on a Service Mesh are traditionally used to protect these vectors, respectively.

An attempt to use one instrument to solve all security tasks is an antipattern. A centralized API Gateway, for example, Spring Cloud Gateway, is ideal for primary traffic termination and filtering. It can offload internal microservices by taking over heavy tasks: receiving JWT tokens from the SPA, verifying their signatures using a cached public key from Keycloak, and enriching the request with HTTP headers identifying the user, for example, X-User-Id. The latency is explained by the need for JSON parsing, token expiration checking, and Redis access for quota verification. Spring Cloud Gateway YAML config is shown in Figure 4.

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri:
https://keycloak.example.com/auth/realms/myrealm/protocol/openid-
connect/certs
  cloud:
    gateway:
      routes:
        - id: business-api
          uri: ${business-svc.url}
          predicates:
            - Path=/api/business/**
          filters:
            - name: RequestRateLimiter
              args:
                key-resolver: "#{@userKeyResolver}"
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
            - name: AddRequestHeader
              args:
                name: X-User-Id
                value: "#{@extractUserIdFromJwt}"
        - id: observability-api
          uri: http://observability-svc:8080
          predicates:
            - Path=/actuator/prometheus
```

Fig. 4. Spring Cloud Gateway YAML config

On the other hand, once the request has passed through the API Gateway, internal traffic between microservices must be protected through mTLS without exception. Custom JWT Filter is shown in Figure 5.

```
public Mono<Void> handle(ServerWebExchange exchange,
GatewayFilterChain chain) {
    String token = extractToken(exchange);
    if (token == null) return unauthorized(exchange);

    // 1. Check local cache for public key
    PublicKey publicKey = cache.get("keycloak_public_key",
        k -> fetchPublicKeyFromJwks());

    // 2. Verify signature and claims
    if (!verifyJwt(token, publicKey)) return unauthorized(exchange);

    // 3. introspection for revocation
    if (needsIntrospection(token)) {
        if (!isTokenActive(token)) return unauthorized(exchange);
    }

    // 4. Add user headers and proceed
    exchange.getRequest().mutate()
        .header("X-User-Id", extractUserId(token))
        .build();
    return chain.filter(exchange);
}
```

Fig. 5. Custom JWT Filter

At the transport level, mTLS delivers superior performance by leveraging persistent TCP connections and hardware-accelerated AES-GCM block encryption in modern processors. Yet the price of this speed is high memory consumption. However, a drawback of this is that the proxy in each pod as a sidecar would require an extra 150 MB in memory (Song et al., 2024). For an enterprise cluster with 1000 pods per cluster, that amounts to 150 GB of extra RAM. There are also resource and financial costs associated with scaling up that need to be accounted for.

mTLS is especially relevant for asynchronous message brokers like Apache Kafka, as failure to enforce transport encryption and authentication could allow rogue application containers in a Kubernetes cluster to capture network traffic or send forged messages to Kafka topics and violate the consistency of the enterprise distribution system. However, Kafka with mTLS and optimal compression settings can have a small impact on overall performance and enable Kafka to deliver messages with high throughput according to empirical testing by Meka (2025).

Secret management is one of the most vulnerable areas in standard Kubernetes installations. Storing DBMS passwords, TLS private keys, and integration tokens in standard Kubernetes Secret objects is a critical security error. By default, these objects are encoded in base64. This means that any process or user with the get secret privilege in a given namespace, or who has obtained access to the etcd node file system, can instantly compromise all credentials.

Zero Trust at the data layer will utilize HashiCorp Vault for a centralized cryptographic key management and secret store. In practice, there are a number of ways to integrate HashiCorp Vault with Kubernetes, namely Vault Agent Injector, Vault CSI Driver, and the External Secrets Operator (Lublinsky et al., 2022).

In contrast to the customary sidecar pattern used by Vault Agent Injector, secrets are mounted directly into the in-memory filesystem of individual pods in a pod's container. Yet this approach places the greatest load on the cluster infrastructure. Each business application requires a separate background agent process.

An alternative, more elegant architectural approach is implemented using the External Secrets Operator. ESO

functions as a single global controller at the scale of the entire cluster. ExternalSecret Resource code is shown in Figure 6.

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: vault-service1
  namespace: my-apps
spec:
  refreshInterval: "60s"         # How often to sync from Vault
  secretStoreRef:
    name: vault-backend-store    # Reference to a SecretStore (or
ClusterSecretStore)
    kind: SecretStore
  target:
    name: vault-service1-creds   # Name of the Kubernetes secret to
create
    creationPolicy: Owner        # ESO will create and manage this
secret
  dataFrom:
    - extract:
        key: dev/service1        # Vault path
```

Fig. 6. ExternalSecret Resource code

The process of synchronizing secrets through the External Secrets Operator is shown in Figure 7.

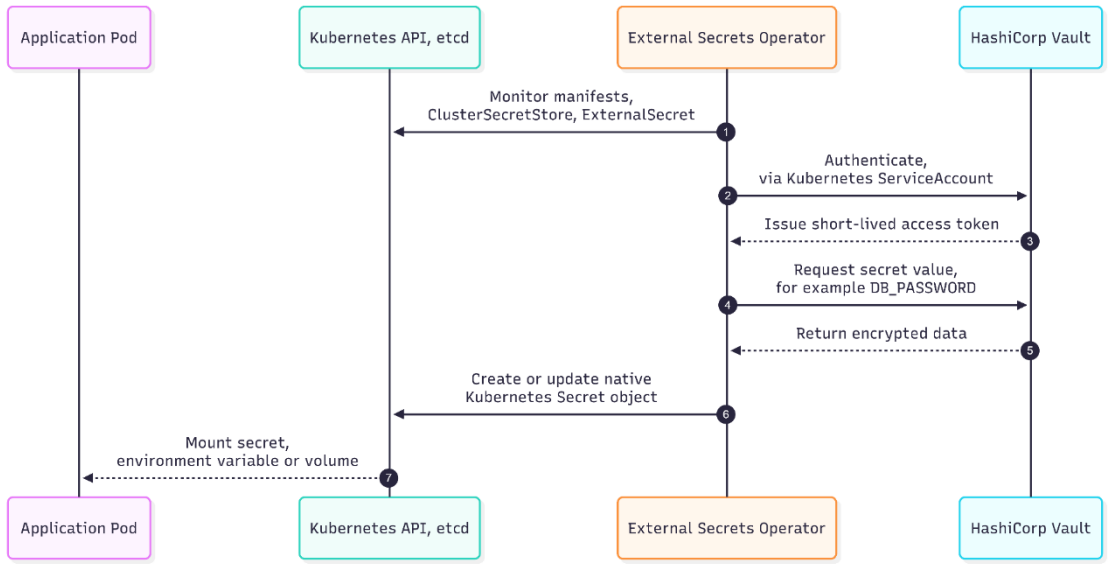

Fig. 7. The process of synchronizing secrets via the External Secrets Operator

As shown in Figure 7, ESO uses the power of custom resources. The operator periodically polls Vault and synchronizes data, creating native K8S Secret objects on the fly. The main architectural advantage of this approach is that when a password is rotated centrally in Vault, ESO automatically updates the corresponding Kubernetes

Secret. This change can be captured by platform tools to initiate a smooth pod restart, allowing the application to pick up the new credentials without manual intervention. This approach reduces system resource consumption. One controller per cluster is used instead of hundreds of sidecars, it also fits the GitOps paradigm with precision, for example, when ArgoCD is used, because ExternalSecret manifests contain only references to Vault paths.

For authentication when microservices access the Kafka broker, Vault serves as a full-fledged public key infrastructure. Dynamic generation of short-lived X.509 certificates improves the cryptographic hygiene of the system. The use of short validity periods eliminates the need to maintain cumbersome certificate revocation lists. Certificates and private keys are delivered to the cluster via ESO, after which a specialized startup script dynamically assembles the protected keystore and truststore files required by Java applications and the native Kafka client.

The critical line of defense is the assurance of integrity, isolation, and versioning at the relational database level. A widespread architectural antipattern is the assignment of database owner privileges to a microservice (Laigner et al., 2021). In the event of a successful compromise of such a service through a remote code execution vulnerability, an attacker can irreversibly modify or destroy the entire table structure, leading to a catastrophic halt of business processes.

Within the proposed architectural model, a contemporary Database DevOps approach is applied using the Liquibase versioning tool. The database schema migration process is physically separated from the application's lifecycle and launched as a one-time task, an init job, or an initContainer before the main microservice starts, while rigorously observing the principle of least privilege.

Liquibase uses a separate, highly privileged set of credentials, dynamically retrieved from Vault, exclusively for applying incremental change files. After successful completion of DDL operations and schema update, the main microservice container starts and authenticates to the database using a completely different set of credentials. The rights of this user are rigidly limited to DML operations such as SELECT, INSERT, UPDATE, and DELETE within specific tables.

Zero downtime schema migration is notoriously difficult and often impossible without write pauses. Where feasible, the Contract/Expand pattern (dual-write, then removal) can be used but only for backward-compatible changes (Malhotra et al., 2024). While database changes require careful migration patterns, canary releases for application code can be handled directly at the API Gateway layer. Using Spring Cloud Gateway, engineers can gradually shift traffic from a stable service version to a canary version by exposing different API routes or by configuring weighted routing. For example, the gateway can route 10% of requests to the canary service and 90% to the stable one, then increment to 40%, 90%, and finally 100%. Such a strategy minimizes the vulnerability window during updates and enables immediate rollback at the routing level simply by changing the gateway's weight configuration or redirecting all traffic back to the stable version if anomalous behavior is detected.

Despite the evident advantages in security and fault tolerance, implementing the described high-technology architecture is associated with several serious operational, organizational, and financial barriers. The analysis enables distinguishing three key groups of risks and limitations in the proposed architectural model.

The orchestration of a distributed Vault cluster, the configuration of seamless integration between Keycloak and corporate Active Directory domains, the management of Kafka broker certificates, and the maintenance of declarative Liquibase pipelines require a deep systemic transformation of engineering culture toward mature DevSecOps practices. The entry threshold for development and infrastructure operations teams is assessed as extremely high. If a service mesh (e.g., Istio) is added for advanced traffic management, engineers may struggle to debug mTLS sessions because network errors and timeouts become obscured by the mesh abstraction.

The introduction of such a complex technological stack in the absence of deep internal expertise leads to critical configuration errors. For example, granting excessive Role-Based Access Control (RBAC) rights to the External Secrets Operator de facto undermines the entire concept of HashiCorp Vault secret isolation. In addition, Vault architecture has a single point of failure: the master key. Although Shamir’s secret-sharing algorithm provides distributed trust, the loss of key fragments or errors in the configuration of Auto-Unseal mechanisms can result in the irreversible loss of access to all cluster

secrets (Ebad & Amara, 2026).

The introduction of sidecar patterns to ensure deep network isolation consumes substantial RAM and requires additional CPU limits. At scale, in large deployments with thousands of containers, the infrastructure overhead of maintaining the security layer can consume up to a quarter of the total budget for cluster compute capacity.

Even so, ignoring these technologies ultimately costs corporations far more. Storing unencrypted secrets in the cluster, weak authentication mechanisms, the use of Implicit Flow in place of PKCE, and the absence of control over inter-service traffic inevitably become entry points for exploitation of zero-day vulnerabilities, automated ransomware attacks, and software supply chain compromise.

4. Conclusion

The design of microservice architectures in the ephemeral Kubernetes environment poses a challenge for system engineers and information security architects. The flexibility, delivery speed, and scalability of the system increase in direct proportion to the growth of the potential attack surface and the complexity of access control logic.

In the course of the conducted study, a reference security model based on the strict principles of Zero Trust Architecture was developed, systematically analyzed, and substantiated. The proposed model achieves all stated objectives.

The problem of vulnerability to authorization code interception in public clients is resolved architecturally by rejecting the outdated Implicit Flow in favor of the current OAuth 2.1 standard, which mandates the use of the PKCE cryptographic mechanism. It has been proven that this ensures reliable session protection with negligible impact on performance.

The centralization of security policies via an API Gateway, in combination with Keycloak and LDAP identity federation, abstracts complex authentication logic away from individual microservices. The use of Redis for caching and Rate Limiting protects the cluster from overload and brute-force attacks at the external perimeter.

A hybrid approach to network protection delivers trust domain separation. The API Gateway handles comprehensive north-south traffic inspection at Layer 7, while mTLS (with certificates issued by Vault PKI) provides robust encryption of east-west inter-service traffic with minimal latency, which is critical for protecting high-load components such as Apache Kafka.

The combination of External Secrets Operator and Liquibase ensures the complete elimination of the hard-coded secrets antipattern. This enforces least privilege at all levels, from secure secret injection into Kubernetes pods to zero-downtime database schema migrations.

The practical significance of this work lies in providing architectural guidance for DevSecOps specialists, IT architects, and cloud platform engineers. Despite the high requirements for personnel engineering qualifications and the inevitable computational overhead, the phased implementation of the examined technology stack is imperative for contemporary enterprise cloud infrastructures. A systemic approach reduces the likelihood of successful cyberattacks and blocks lateral movement of attackers within a compromised cluster, while also providing a robust, scalable foundation for automated information security audits amid continuous tightening of global regulatory standards.

References

1. Cesarano, C., & Natella, R. (2025). KubeFence: Security Hardening of the Kubernetes Attack Surface. *ArXiv*, 497–510. https://doi.org/10.1109/dsn64029.2025.00054
2. Chen, Q., Liu, Y., Tan, R., Jin, Z., Xiao, J., Wang, X., Zhang, F., & Liu, Q. (2025). Shadowkube: enhancing Kubernetes security with behavioral monitoring and honeypot integration. *Cybersecurity*, *8*(1). https://doi.org/10.1186/s42400-025-00372-7
3. Chmelev, A. (2025). Evolution of User Session Security Using OAuth 2.1 and Openid Connect: 2025 Practices. *Universum:Technical Sciences*, *134*(5). https://doi.org/10.32743/unitech.2025.134.5.20012
4. Ebad, S. A., & Amara, M. (2026). The Principle of Least Privilege in Microservices: A Systematic Mapping Study. *Applied Sciences*, *16*(3), 1495. https://doi.org/10.3390/app16031495
5. Faustino, D., Gonçalves, N., Portela, M., & Rito Silva, A. (2024). Stepwise migration of a monolith

to a microservice architecture: Performance and migration effort evaluation. *Performance Evaluation*, *164*, 102411. https://doi.org/10.1016/j.peva.2024.102411

6. Hardt, D., Parecki, A., & Torsten Lodderstedt. (2024, January 9). *The OAuth 2.1 Authorization Framework*. IETF Datatracker. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-10
7. Hussain, A., Aziz, A., Syed, H. J., & Raza, S. (2025). Preventing IP Spoofing in Kubernetes Using eBPF. *Computers, Materials & Continua*, *84*(2), 3105–3124. https://doi.org/10.32604/cmc.2025.062628
8. Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *ArXiv*. https://doi.org/10.48550/arXiv.2103.00170
9. Lublinsky, B., Jennings, E., & Spišaková, V. (2022). A Kubernetes Bridge operator between cloud and external resources. *ArXiv*. https://arxiv.org/abs/2207.02531
10. Malhotra, A., Elsayed, A., Torres, R., & Venkatraman, S. (2024). Evaluate Canary Deployment Techniques using Kubernetes, Istio and Liquibase for Cloud Native Enterprise Applications to Achieve Zero Downtime for Continuous Deployments. *IEEE Access*, *99*. https://doi.org/10.1109/access.2024.3416087
11. Meka, J. (2025). Financial Services Cloud Transformation: Securing Sensitive Data in Kafka Event Streams. *Journal of Computer Science and Technology Studies*, *7*(4), 1023–1028. https://doi.org/10.32996/jcsts.2025.7.4.115
12. Nascimento, B., Santos, R., Henriques, J., Bernardo, M. V., & Caldeira, F. (2024). Availability, Scalability, and Security in the Migration from Container-Based to Cloud-Native Applications. *Computers*, *13*(8), 192. https://doi.org/10.3390/computers13080192
13. Rahat, T. A., Feng, Y., & Tian, Y. (2022). Cerberus. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3548606.3559381
14. Saleh, S. M., Madhavji, N. H., & Steinbacher, J. (2025). Systematic Review of Identity-Centric Security in Cloud-Native CI/CD Pipelines. *Proceedings of the 2025 10th International Conference on Cloud Computing and Internet of Things*, 23–32. https://doi.org/10.1145/3785520.3785525
15. Singh, J., & Chaudhary, N. K. (2023). Unified Singular Protocol Flow for OAuth (USPFO) Ecosystem. *ArXiv*. https://doi.org/10.48550/arXiv.2301.12496
16. Song, E., Song, Y., Lu, C., Pan, T., Zhang, S., Lu, J., Zhao, J., Wang, X., Wu, X., Gao, M., Li, Z., Fang, Z., Lyu, B., Zhang, P., Wen, R., Yi, L., Zong, Z., & Zhu, S. (2024). Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. *Proceedings of the ACM SIGCOMM 2024 Conference*, 860–875. https://doi.org/10.1145/3651890.3672221
17. Xi, N., Liu, J., Li, Y., & Qin, B. (2023). Decentralized access control for secure microservices cooperation with blockchain. *ISA Transactions*, *141*, 44–51.