Manuscript Received: 24 Sep. 2025, Revised: 29 Oct. 2025, Accepted: 11 Nov. 2025

AN IN-DEPTH REVIEW OF AUTHENTICATION MECHANISMS IN MICROSERVICE ARCHITECTURES

*Md. Abdul Momin[1], M.M. Musharaf Hussain[2], and Md. Ezharul Islam[3]
[1]Department of Computer Science Engineering, Jahangirnagar University
[2]Department of Computer Science Engineering, Jahangirnagar University
[3]Department of Computer Science Engineering, Jahangirnagar University
Email: momin99cse@gmail.com

Abstract: Authentication in microservice architectures (MSA) is hard because services are spread out. Managing user identities is tricky. Keeping communication between services secure is important. Controlling access to APIs is also complex. It is more challenging than in traditional monolithic systems. The main goal of this study is to review and compare existing authentication mechanisms for microservices to identify effective and scalable solutions. This review uses a structured narrative method, analyzing recent research and technologies such as JSON Web Tokens (JWT), session-based authentication, Single Sign-On (SSO), passwordless, and biometric approaches. The findings show that token-based methods like JWT improve scalability and user experience but are vulnerable to token theft. Session-based systems offer stronger central control but struggle to scale in large networks. Passwordless and SSO solutions enhance usability but still require strong security controls. This review compares different authentication methods clearly. It describes the pros and cons of each system. It also explains how new trends like zero-trust, adaptive, and decentralized authentication may change the future. These trends help make microservice systems more secure and scalable.

Keywords: *Cookie-Based, OAuth 2.0, Session-Based, Passwordless, JSON Web Token, Biometric.*

1. INTRODUCTION

1.1 Background

Microservice architecture is among the fastest-growing architectural paradigms in commercial computing today. First introduced in a white paper by Martin Fowler and James Lewis, it has since emerged as a de facto standard for developing large-scale commercial applications [1]. Microservices are essentially small, independent programs that talk to each other using messages. So, a microservice architecture is basically a big application made up entirely of these little services [2]. It's a way of designing software where an application is split into many separate services, with each one handling a specific part of the business. Moving from an older, all-in-one "monolithic" system to this microservice approach offers many benefits. Still, as a distributed architecture, the microservice architecture has a larger attack surface, making it more challenging to share user context. Therefore, different authentication services are needed under the microservice architecture to respond to more significant security challenges.

1.2 Importance of Authentication

Securing microservices can be challenging since each service requires individual attention, unlike monolithic architectures, where security strategies apply to a single app. There is also limited guidance on improving security in service-oriented architectures. Authentication (verifying identity) and authorization (granting resource access) are vital regardless of architecture. Protocols like OAuth 2.0 for delegated authorization and OpenID Connect for adding authentication to OAuth 2.0 are key to addressing these concerns [3]. Research shows that while microservices often rely on inherent trust, some adopt a "zero-trust" approach, where trust is continuously assessed. Neglecting authentication and authorization in any microservice can compromise the entire system. Thus, security studies should prioritize these aspects. Recognizing the critical role of authentication for microservices security, this paper explores the mechanisms that provide robust identity verification and secure access control in distributed systems. Given the critical need for robust, scalable, and user-friendly authentication solutions in the face of these inherent MSA challenges, this paper proceeds to analyze various authentication methods and their applicability within these distributed environments.

1.3 Scope

This paper provides a comprehensive investigation of authentication mechanisms in microservice architectures, emphasizing their critical role in secure communication and data integrity while addressing unique challenges such as decentralized identity management, stateless communication, and scalable security solutions. It evaluates methods like Single Sign-On (SSO), biometrics, JWT, and passwordless authentication. It analyzes their trade-offs in stateless token-based systems versus session-based overhead and third-party risks in protocols like OpenID Connect. We looked at the big challenges with microservices: services not remembering things, hitting scaling limits, security risks between services, more places for attacks, and making them user-friendly. We found solutions like API gateways, service meshes, and OAuth 2.0/OIDC. We also looked at real-world examples and new ideas like zero-trust, adaptive authentication, and decentralized identity to predict future trends. Our goal is to create a system that balances security, growth, and ease of use, while identifying areas for further study.

1.4 Review Methodology

This review followed a simple and clear process to study different authentication methods in microservice systems. Research papers were taken from IEEE, ACM, Springer, and Google Scholar. The study covered works published between 2010 and 2024.The focus was to learn how token-based, session-based, and passwordless methods help secure microservices. Only papers related to authentication in microservices were used. Each method was tested for security, speed, and scalability. We also checked how easy it is to use. The results show which methods work best. New ideas like zero-trust and adaptive authentication make microservices safer and smarter.

2. MICROSERVICE AUTHENTICATION FUNDAMENTALS

2.1 Key Concept

Authentication is verifying the identity of users or entities within an access control system, while authorization determines what authenticated users are permitted to access. For instance, authentication validates your identity, such as logging in with a username and password, whereas authorization governs access to specific resources, such as granting administrative privileges. In microservice architectures, where monolithic tasks are decomposed into multiple independent services, authentication becomes more complex due to the dynamic and distributed nature of access points. This shift introduces new security and operational challenges [4].

Authentication in microservices typically occurs in three contexts:

- Authenticating end-users who access the application.
- Authenticating internal communication between microservices.
- Authenticating external services that connect to microservices via APIs.

One common approach is to have each service perform authentication independently. However, this can result in redundant code and inconsistent security enforcement. A more efficient and scalable solution is to implement a centralized authentication service. This allows microservices to delegate user verification, focusing solely on their specific business logic and improving overall system maintainability [5]. A sample authentication flow within a microservice architecture is shown in Figure 1.

2.2 Authentication vs Authorization

Authentication is how a system checks that you're you, whether that's with a password, your fingerprint, face scan, or an extra step like two-factor authentication (2FA). Its main job is making sure only the right people get in. Once you're in, authorization takes over. Once you're in, it controls what you can do like looking at files, changing data, or opening specific apps. Your access depends on your role (RBAC) or your characteristics (ABAC). This process verifies 'who you are?' so the user needs to supply login details to authenticate. Authorization is the process of verifying if the authenticated user is authorized to access specific information or be allowed to execute a certain operation. This process determines which permissions the user has. Figure 2 shows a basic representation of distributed authentication in a microservice architecture, where each service may handle its authentication against a central database.

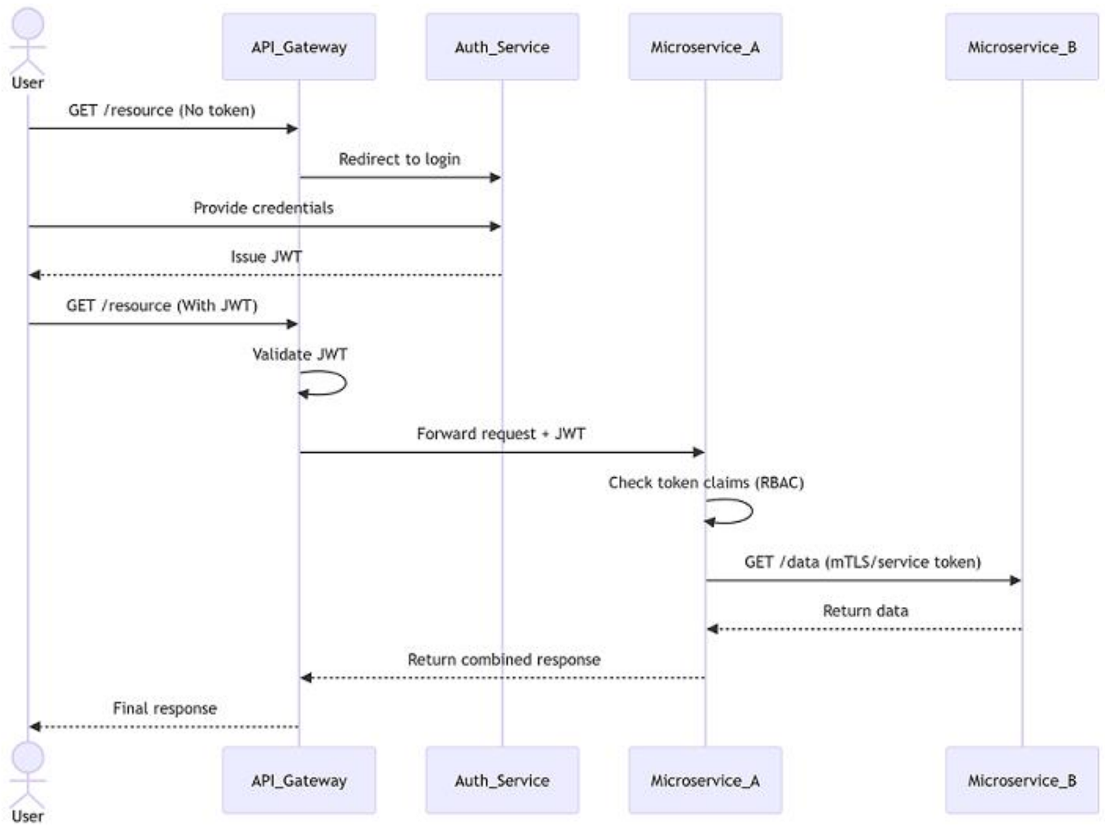

Fig. 1. *End-to-end authentication flow showing token propagation and inter-service mTLS*

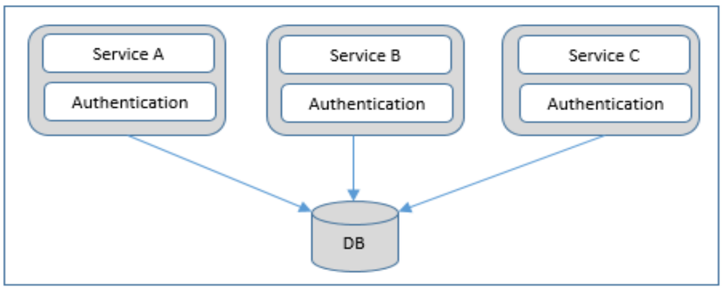

Fig. 2. A simplified representation of authentication flow in a microservice ecosystem, showing service-specific checks against a central authority.

2.3 Key Feature

Microservices decompose a large system into small, independent services. Based on a specific task or business capability, each service is developed. This service can be independently developed and deployed. This improves flexibility and scalability in development and deployment. Microservices are language-independent and operate within a defined domain, confirming clarity and specialization [6]. Organizations like Amazon, eBay, and Netflix use the Microservice architecture pattern rather than a single, monolithic application. These characteristics of microservices, loose coupling, independent deployment, and language neutrality, introduce specific challenges for authentication.

2.4 Authentication Challenge

Delegated authorization and shared authentication are an essential part of modern web security. The most tangible and visible mechanism is social login, which many web services support today. However, delegated authorization and shared authentication protocols have many interesting applications for securing inter-service communication in microservice architectures [7]. As MSA and distributed architectures evolve, security has become increasingly important. Each microservice introduces a new entry point for both internal and external access, creating numerous security challenges that need to be addressed [8]. Microservices improve scalability and efficiency by being small, easy to develop, and operate. They simplify continuous delivery, making it easier to add new services or upgrade existing ones. By focusing on specific tasks, microservices reduce dependencies and minimize errors, making maintenance simpler. Independent services with their databases can be reused without deep internal knowledge, and teams can choose the best technology for each service [9]. Microservices bring big advantages but make security trickier. Since they're spread out, each service needs its own login checks, which can get messy. Without a single control point, keeping access secure is harder; hackers have more ways in. To stay safe, every service must tightly manage who gets in and how they talk to others**.** Authentication in distributed systems presents several challenges that require careful management. Microservices make security harder in a few ways:

- No Memory: Each request is separate, so tracking logins gets tricky.
- Growth Issues: More users and services can overload a central login system.
- Service-to-Service Trust: Every microservice must securely verify others.
- More Hacking Risks: Extra entry points mean more weak spots.
- User Hassles: Strong security may slow down or be annoying for logins.

In summary, microservice architectures demand robust, distributed authentication solutions to address the inherent challenges of scalability, security, and seamless interoperability while maintaining stringent access control.

3. EXISTING AUTHENTICATION APPROACH

3.1 Token-Based Approach

Token-based authentication is a modern security mechanism where clients exchange credentials for a server-generated token, which is then included in HTTP headers for subsequent resource requests [10]. These system-generated constructs serve as signed (though not encrypted) identifiers containing user authentication data, representing a significant evolution from traditional password-based systems. Tokens can be implemented as both hardware devices (like USB keys or smartcards) and software-based solutions, often incorporating multi-factor authentication. RFC standards define several functional token types: perishable tokens for single actions, session tokens for multi-use within a session, reusable access tokens, and single-use refresh tokens [11]. Physically, tokens manifest in three forms: connected (e.g., USB devices requiring physical contact), contactless (e.g., Microsoft's "magic ring" using proximity communication), and disconnected (e.g., smartphones authenticating remotely) - all requiring initial user verification like password entry before granting access [12].

The approach offers notable advantages over traditional methods. Token systems enhance security through self-contained validation restricted to the issuing server, while providing administrators with granular control over expiration and permissions across platforms. User experience improves through streamlined authentication without additional hardware requirements. However, challenges exist, particularly around token theft risks, where compromised tokens enable impersonation attacks. To keep tokens secure, must have a good system for renewing or revoking expired ones. Also, store them safely on the client side to prevent XSS attacks. Nowadays, many apps and websites use tokens like JWTs, OAuth 2.0, OpenID, and others (such as Hawk or Firebase tokens). These options make token-based security both flexible and always evolving. Let's explain one by one.

3.1.1 JSON Web Tokens (JWT)

JWTs let users log in without storing sessions on the server. They're tiny, hold user data, and work well across multiple services. A JWT has three parts: header (how it's secured), payload (user info), and signature (to check if it's valid). No server storage means faster performance for big systems. The tokens support both signing (JWS) and encryption (JWE), with signed JWTs using HMAC or public/private key pairs (RSA/ECDSA) to verify integrity without necessarily ensuring confidentiality [6][13][15]. JWTs work great for microservices because each service can check the token on its own—no need to hit the database. This saves memory and CPU, making scaling easier. Their small size and HTTP-friendly format also make them good for Single Sign-On (SSO) and OAuth flows [13][15]. But there are downsides: once issued, JWTs can't easily be revoked before they expire, so you have to manage their lifetime carefully. Also, some other risks like stolen signing keys, larger payloads than session tokens, and tricky crypto setup [14][16][17]. For top security, use short-lived JWTs, avoid storing sensitive data in them, and add extra defenses like token binding and regular key rotation.

3.1.2 Session Token

Session tokens represent a stateful approach to authentication that contrasts sharply with stateless JWT implementations. Unlike self-contained JWTs, session tokens rely on server-side session tracking, where a digitally signed identifier is created upon login and stored either in server memory or a database. the client also maintains the session ID in a cookie [18][19]. This method doesn't scale well in distributed systems. Each server has to sync session data, which eats up memory and can cause sync issues. If the server goes down, sessions get wiped out. Also, cookies don't work smoothly with APIs or mobile apps. While session tokens solve some JWT problems (like being unchangeable), they can open the door to CSRF and session hijacking—unless you lock down cookie settings. Because of this, session tokens work best in single-server setups or cases where tight session control matters more than easy scaling, unlike stateless JWTs.

3.1.3 OpenID Connect

OpenID Connect (OIDC) is now the most popular way to handle single sign-on (SSO). It takes OAuth 2.0, which normally just handles permissions, and adds a way to prove who users are. OIDC does this by using special ID tokens (in JWT format) while still working with OAuth's existing login process [20][21]. This dual functionality powers ubiquitous "Login with Google/Facebook" implementations, allowing users to authenticate once with a trusted identity provider (like Microsoft or Amazon) and then access multiple services without re-entering credentials - a capability that has made OIDC the foundation for SSO across over a million websites [10]. The user, the app you're using, and the login provider (like Google or Facebook) are the three main actors of OIDC. By working together, they confirm identity in OAuth. Short-lived tokens mean less risk if they're stolen. Social logins make signing in easier. And you benefit from the strong security of big tech companies. But there are downsides too. Since OIDC runs on OAuth 2.0, it can have the same security holes, especially if tokens aren't protected properly or if setups are misconfigured. Different providers might implement things slightly differently, and if a major login service has issues, it can break access for tons of apps at once. Despite these limitations, OIDC's combination of OAuth's flexible authorization with standardized identity verification has made it the preferred choice for enterprise SSO implementations and API security, particularly in scenarios requiring integration with multiple identity providers while maintaining centralized control over authentication policies [10][20][21].

3.2 Session-Based Approach

Session-based authentication represents a traditional and widely used security mechanism. In this type of authentication, the server creates and maintains session records of authenticated users. Unlike token-based approaches, this method depends on server to maintain the authentication continuity user. When a user successfully logs in, the server generates a unique session identifier (SID). This SID contains the information about the user, expiration details, and stores this in the server, and conveys only the session ID to the client via a cookie. This cookie is automatically included in subsequent HTTP requests, allowing the server to validate the session and authorize access [22][23][24]. The complete authentication process involves credential verification, session creation, request validation, and session termination upon logout [25]. Session-based login has some nice benefits—it lets servers fully control active sessions, keeps sensitive user data off devices, and works smoothly with older systems. But it struggles in today's cloud-based apps. Storing sessions on the server can slow the system when more users log in. By checking these sessions adds extra workload. There are also security risks, such as session theft, if cookies aren't locked down with protections like HttpOnly and SameSite tags. Developers can use different approaches, from simple cookies to storing sessions in databases or fast systems like Redis. Big companies often add Single Sign-On (SSO) to allow users to access multiple services with a single login. Sessions work fine with basic or older apps, but many newer systems, like microservices, prefer token-based logins. Finally, session logins can work well, but only if you set up strong security and have the right infrastructure to handle sessions across multiple servers.

3.2.1 Cookie-Based

In microservice setups, cookie-based authentication uses HTTP cookies to keep track of user sessions, helping overcome HTTP's stateless nature by storing authentication details on the client side [26]. This allows users to stay logged in across multiple requests without re-entering credentials, as servers check the session ID stored in the cookie to verify identity [27]. While cookies work well in some cases—thanks to browser support, easier session handling, and customizable expiration—they also create problems in distributed systems. Since microservices are designed to be stateless, cookies (which rely on state) don't fit perfectly. They also bring security risks like XSS and CSRF attacks, especially when services communicate with each other [26]. Another issue is that cookies depend on browsers, making them less useful for API-driven microservices or mobile apps. Plus, sending cookies with every request can slow down performance in high-traffic systems [27]. Security measures like

Secure, HttpOnly, and SameSite flags can reduce risks, but they don't completely fix cookies' drawbacks for modern, distributed microservices.

3.2.2 Server-Side Session Storage

In microservice architectures, server-side session storage keeps user session data on the server while using cookies to track session IDs [28]. In this setup, the server controls all session data. Admins can quickly terminate sessions when required, and user data remains more secure as it's stored server-side rather than on personal devices. Though effective for conventional applications, this approach runs into challenges with microservices. The central session store becomes a bottleneck—if it goes down, authentication fails across all services [28]. Performance also takes a hit because every service has to check sessions by making remote calls to the central database, adding delays. On top of that, microservices work best when they're stateless—server-side sessions break that rule. Tools like Redis can help handle more traffic, but they add extra setup and complexity. And for APIs (not just websites), sharing session IDs across different services gets complicated fast. While some tools make it easier to set up, the clash between centralized sessions and decentralized microservices often makes token-based authentication a better fit for modern systems.

3.2.3 Single Sign On (SSO)

Single Sign-On (SSO) simplifies logging in for microservices by letting users sign in once to access all connected services [29][30]. Instead of multiple passwords, a central login system (using standards like SAML or OAuth) verifies users and provides secure access tokens [31]. This approach brings several benefits: easier account management since credentials are controlled in one place, better security with features like two-factor authentication [32], and less IT workload for password resets and access management [33]. However, SSO introduces specific challenges in microservice environments. The centralized IdP becomes a critical single point of failure—any outage can disrupt access to all dependent services [19]. Security risks are amplified since a compromised IdP could potentially expose the entire microservice architecture [32]. Setting up SSO adds complexity because every service must validate tokens the same way while still handling its own access rules [32]. It also works better for web services than backend systems, often needing extra security steps for service-to-service communication. While SSO makes logins much easier, companies need to weigh these benefits against creating a central point in what should be decentralized systems. They must plan for backups, strong token checks, and backup plans to keep everything running smoothly [29][33].

3.3 Passwordless Authentication Approach

Passwordless login lets you sign in without passwords. Your device creates a secure key tied to your face, fingerprint, or PIN. When logging in, you just prove it's you - no password needed. The system checks your unique key instead. Some even use your location or habits for extra security. It's like a digital lock that only you can open. Now, understand the difference between passwordless login and MFA. MFA requires multiple steps to log in, like a password plus your phone or fingerprint. Passwordless login removes passwords completely. It only uses secure methods like your face scan or a security key. This makes logging in both safer and easier. Regular MFA still needs passwords, which hackers can steal. Passwordless MFA is different - it might combine your fingerprint with a security token, but it never uses passwords. This gives extra protection without the risks. Passwordless login has big benefits. First, it's more secure. No passwords means no weak passwords or password leaks. Second, it's easier for users. No more remembering or resetting passwords. Third, it helps meet security rules [34]. But there are downsides too. Setting it up can cost money. You might need special hardware like security keys. People may need training to use it. If you lose your security key or phone, getting back in can be hard [34]. Various passwordless authentication methods are used today, including Email Link Authentication, SMS One-Time Passwords (OTP), Biometric Authentication (Fingerprint/Face ID), Push Notification Authentication, Magic Link Authentication, Hardware Tokens (e.g., YubiKey), WebAuthn, QR Code Authentication, OAuth-based Social Logins, and Authenticator Apps like Google Authenticator. Among these, we will focus on three major approaches in the following sections [34].

3.3.1 Biometric

Biometric authentication uses unique physical characteristics like your fingerprint, face, or eye pattern to verify a user. The system works by comparing your live scan data with a securely stored version data. If they match, you get access. This technology is now everywhere: from unlocking your phone to airport security checks, making it both convenient and highly secure [34]. In microservices systems, biometric login makes security stronger while keeping things easy for users. Since microservices work as separate parts talking over networks, safe identity checks are vital. Biometrics solve this by replacing hackable passwords with your fingerprint or face scan - no more remembering or sharing credentials across services. This works perfectly with modern access control systems, giving centralized security without a password. Biometrics like fingerprint scanning are widely used and also convenient, but not always foolproof. Factors like wet, dirty, or injured fingers can cause recognition errors [34]. That's why having backup login options is essential. Even with its limits, biometrics are still a top choice for passwordless security. They offer the right mix of strong protection, easy use, and scalability that microservices need to work smoothly [34].

3.3.2 SMS One-Time Password (OTP)

SMS-based One-Time Password (OTP) is a simple way to keep accounts safe. It sends a short code to your phone when you log in or make a transaction. You have to enter that code to prove it's you. This adds extra security beyond just using a password. The code works only once and for a short time. It's easy to use because almost everyone has a phone that can get texts. The idea of OTP was first introduced by Leslie Lamport in the 1980s. Now, it's very common because it's fast, easy, and works anywhere [34]. In a microservices system, many small services operate independently. They need to talk to each other in a secure way. SMS OTP (One-Time Password) is a simple method for this. It checks if you're real without saving your password, so hackers can't steal it. One main service can handle all the OTP checks. This keeps the security the same across all services. SMS OTPs are widely used, but they have some problems in high-security systems. They need a mobile network to work. If the network is weak or down, the OTP might not arrive. They are also at risk from SIM-swapping attacks. Hackers can steal the OTP if they control the SIM. SMS messages can also be intercepted by attackers. If the user loses their phone, it can stop the login process. Still, SMS OTPs are easy to set up and simple for users. That's why they are still useful. When used with other security tools, they can work well in microservices [34].

3.3.3 Social Login

Social login agreements you sign into apps or websites using your Facebook, Google, or Twitter account. No need to remember another password. This makes signing up faster and easier [34][40]. Since these accounts are already verified, logins are more secure, with fewer failed attempts. For apps with multiple services (microservices), social login acts as a central ID checker. Instead of handling logins themselves, these services trust platforms like Google to verify users securely. This keeps things simple and safe. But there are downsides. If Facebook or Google has an outage, users might get locked out. Also, social profiles sometimes have old or missing info, which can cause problems. Users may also forget which social platform they used to log in, complicating account recovery [34]. Despite these challenges, social login remains a popular and effective authentication method for microservices-based applications seeking to streamline user experiences while offloading authentication complexity to external providers [34][40]. Sections 3.1, 3.2, and 3.3 compare different authentication methods. JWTs are good for scalability and don't need a server to store session data. But session tokens give more control, although they may slow things down in large systems. Passwordless and biometric methods are safer and easier for users. Still, they come with problems like device risks and handling sensitive data. These new methods are quite different from token or session systems. So, it's important to choose the method that works best for the system and can deal with any risks. How these methods are set up and managed also depends a lot on the design of the microservice system, which we'll look at in the next section.

4. ARCHITECTURAL PATTERN FOR AUTHENTICATION

The API Gateway and Service Mesh are architectural patterns for managing authentication in microservices. The API Gateway centralizes client authentication, handling tasks like validation and traffic management, often using JWTs. It forwards secure requests to downstream services. The Service Mesh handles communication between your internal services. It uses sidecar proxies to encrypt traffic (with mTLS), verify identities, and control access. Meanwhile, the API Gateway deals with outside requests from clients. Together, they keep your microservices secure both inside and out while making your system more scalable. Here's the reorganized and concise comparison of API Gateway and Service Mesh Architectural Patterns for Authentication based on the provided references.

TABLE 1. Comparison of API Gateway and Service Mesh Architectural Patterns

Aspect	API Gateway	Service Mesh
Definition	An API Gateway is a central entry point in microservice architectures, streamlining communication between clients and services. It manages authentication, load balancing, caching, and monitoring, aligning with design concepts like the Facade pattern. This centralized reverse proxy simplifies client interactions and enforces runtime policies while abstracting service complexities [41].	A service mesh is an infrastructure layer managing inter-service communication in microservices. It uses lightweight proxies (sidecars) to handle traffic monitoring, control, and security. This layer enables encryption, access control, and observability, reducing service code modification needs while improving communication efficiency [45].
Authentication Mechanism	Implements client-to-service authentication by validating access tokens (e.g., JWTs) issued by identity providers. Tokens include user claims and are forwarded to downstream services for fine-grained access control. This federated identity approach decouples authentication from service operations [41].	Handles service-to-service authentication using mTLS (mutual TLS) for encryption and identity verification. Authentication and access control policies are applied at the sidecar proxy level, ensuring secure internal communication without modifying the service code [45].
Security Features	Centralized SSL/TLS encryption prevents man-in-the-middle attacks. Tokens validated at the gateway ensure secure external access. Hosting microservices on private subnets and whitelisting gateway IPs enhances security. Tools like Let's Encrypt and AWS Certificate Manager simplify certificate management [42].	Built-in mTLS ensures encrypted communication between services. Advanced service mesh frameworks like Istio and Consul support features like role-based access control (RBAC), automatic certificate management, and integration with tools like HashiCorp Vault for secure certificate storage and rotation [45].
Challenges	Single Point of Failure: Gateway downtime disrupts system access. Scalability: This can become a bottleneck under high traffic. Performance Overhead: Centralized request validation adds latency. Complexity: Managing policies for numerous services is challenging. Security Risks: Gateway is a prime target for DDoS and injection attacks [43][44].	Scalability: While frameworks like Istio support multi-datacenter setups and load balancing, they add complexity. Configuration Overhead: Requires knowledge of sidecars, control planes, and service intentions. Latency: Proxies in every service introduce latency. Framework-Specific Limitations: Simpler solutions like Linkerd lack advanced automation compared to Istio or Consul [46].
Scalability and Automation	Techniques like load balancing, retries, and circuit breakers improve scalability but add operational complexity [44].	Efficiently scales with proxies; Istio and Consul offer advanced automation for load balancing and access control, while Linkerd emphasizes simplicity with built-in mTLS and traffic routing for smaller setups [46].
Example Frameworks	AWS API Gateway, Kong, Apigee	Envoy (data plane), Istio (advanced framework), Consul (multi-datacenter), and Linkerd (lightweight security) [45][46].
Best Use Case	Ideal for client-facing authentication, offering centralized control, API management, and backward compatibility for dynamic microservices [41].	Suited for securing service-to-service communication in microservices, especially in distributed or zero-trust architectures [45].

5. CHALLENGES AND CONSIDERATIONS AND COMPARATIVE ANALYSIS OF AUTHENTICATION MECHANISMS

Because microservices are spread out, they have more chances to be attacked, especially when used in the cloud with things like virtualization, automated tools, and third-party software [47]. Some big risks in authentication are weak methods like basic login or hardcoded passwords, poor handling of tokens, and secrets that get exposed. Other issues involve token replay attacks, improper RBAC, insecure communication, and vulnerabilities in API gateways. Centralizing authentication reduces inconsistencies but can introduce latency and scalability challenges. Stateless token-based methods like JWTs reduce reliance on centralized storage, while session-based and passwordless methods add complexity. Managing authentication in microservices is complex due to distributed systems, scaling, and compliance. Solutions like API gateways, OAuth, and strong token management help solve these problems, but need to be set up carefully. Here, we compare different ways to handle authentication in microservice architecture, looking at their security risks, speed, scalability, and how complex they are.

5.1 Comparative Analysis

Table 2 compares the authentication methods. JWT is fast and scalable, but needs careful token management. Server-side sessions allow quick revocation but can slow the system and add state. Actually, there is no perfect solution. The choice depends on security, performance, and complexity needs.

TABLE 2. Comparison of various authentication approaches of the Microservice architecture

Method	Security Risks	Performance	Scalability	Complexity
JSON Web Token (JWT)	Susceptible to token theft, replay attacks, or misuse if not secured properly (e.g., via HTTPS). Long-lived tokens are risky.	Lightweight; performs well as tokens are self-contained (no DB lookup needed).	Highly scalable, as tokens are stateless and don't require server-side storage.	Moderate; requires proper implementation and understanding of signing, encryption, and expiry handling.
Session Token	Vulnerable to session hijacking or fixation if tokens aren't secured properly.	Requires server-side storage, which can increase latency with large user bases.	Scalability is limited by server-side storage requirements, but load balancers and replication can help.	Simple to moderate, depending on how session storage is managed (e.g., in-memory vs. distributed).
OpenID Connect	Relies on third-party identity providers, making it dependent on their security practices.	Performance depends on the external provider; generally efficient.	Scales well, as identity delegation reduces backend storage requirements.	High: involves integration with identity providers, token validation, and potential reliance on external services.
Cookie-Based	Susceptible to cross-site scripting (XSS) or cross-site request forgery (CSRF) attacks if not mitigated.	Efficient, as cookies are sent with each request, but can add overhead for large cookies.	Efficient, as cookies are sent with each request, but can add overhead for large cookies.	Simple to moderate; depends on implementing secure cookie attributes (e.g., HTTP Only, Secure, SameSite).
Server-Side Session Storage	Risk of session hijacking; requires secure management of session identifiers.	Slower than JWTs due to server-side lookups, especially for distributed systems.	Scalability is limited by the performance of the session storage infrastructure.	Simple to moderate; session management libraries and distributed systems increase complexity.
Single Sign-On (SSO)	Centralized authentication poses a single point of	Efficient for users as authentication is done once, but	Scales well across services but depends on the	High; requires integration across multiple services,

	failure; breaches can compromise multiple services.	latency can increase with more services connected.	underlying identity provider infrastructure.	including token handling and permissions management.
Biometric	Potentially insecure if biometric data is stored improperly; spoofing or replay attacks are possible.	Fast verification once data is collected, but collection hardware can slow the process.	Scalable, but limited by hardware availability and compatibility across devices.	High; requires specialized hardware, software integration, and secure handling of biometric data.
SMS One-Time Password (OTP)	Susceptible to SIM swapping, interception, or phishing attacks.	Susceptible to SIM swapping, interception, or phishing attacks.	Limited by reliance on third-party SMS gateways and phone carrier network scalability.	Moderate; implementation is simple, but reliance on external SMS services adds dependency.
Social Login	Relies on third-party providers; breach or API changes can impact functionality.	Fast, as user information is retrieved from the provider; dependent on the provider's performance.	Scales well since identity management is offloaded to the provider.	Moderate to high; involves integration with multiple social providers and handling provider-specific APIs and tokens.

6. EMERGING TRENDS AND FUTURE DIRECTIONS

6.1 Zero Trust Architecture

Zero Trust (ZT) is a modern cybersecurity approach that follows the idea of "never trust, always verify." It assumes threats can come from anywhere—inside or outside the system—so nothing is trusted by default. Zero Trust Architecture (ZTA) is the framework used to apply this model. Zero Trust makes sure that users only get the access they really need. It keeps checking who they are again and again to stay safe. The system works like an ultra-careful security guard - it gives people the bare minimum access they need, constantly checks their identity (using multiple verification steps), and monitors everything nonstop to spot and stop threats immediately. This method works well in today's cloud systems, where many people work remotely and use mobile devices or IoT, which makes traditional security methods less useful. ZT works by assuming that breaches can happen anytime, so it controls every access request carefully. The system constantly adapts security based on who's accessing what and from where. In modern apps, it safely connects all the moving parts behind the scenes using bulletproof verification methods. It also uses detailed access rules with methods like Attribute-Based Access Control (ABAC) and tools like Open Policy Agent (OPA). Managing user identity becomes easier with systems like IAM and Federated Identity Management (FIM). Tools like Istio help by watching the system, finding any unusual activity, and applying rules automatically to keep everything secure. Shifting to Zero Trust takes time and planning. It involves checking for risks, adding new tools step by step, and finding a good balance between strong security, ease of use, and cost [47].

6.2 Adaptive and Contextual Authentication

An adaptive system has managed resources and adaptation logic. It uses a feedback loop—Monitor, Analyze, Plan, and Execute—all working with a shared knowledge base. It helps you understand what needs to change, why, and how to adjust—automatically tightening security if something seems off, but staying smooth when all looks good. It checks your usual patterns—like where you are, what device you use, and when—to decide whether to ramp up protection or keep access easy.New methods like checking user behavior, always-on authentication, and AI-based threat detection are part of this. Also, passwordless options like biometrics and decentralized IDs are becoming popular. They make logins more convenient and private while still following security rules. These systems work with zero-trust and IoT, making security smarter and more user-friendly [48][49].

6.3 Decentralized Authentication System

Most old security systems depend on a central server or a trusted third party. Blockchain is different because it is fully decentralized, very secure, and transparent. One simple way to use it is by storing each device's ID and key pair on the blockchain. This lets devices be managed using asymmetric cryptography. This method is strong and secure, but it's not good for small sensors because asymmetric cryptography needs a lot of processing power [50]. Decentralized login systems (like those using blockchain) are a smarter, safer way to handle user authentication across apps and services. Instead of relying on a single company or server to verify your identity, which can slow things down or become a hacking target, it lets devices confirm who you are directly, without a middleman.

7. CONCLUSION

7.1 Summary and Recommendations

In microservices, authentication is not just for users. Services also need to verify each other, and external APIs must be secure. Using a separate authentication service helps the system scale and run efficiently. Microservices have challenges. They need to keep communication stateless, stay secure, scale well, and give a good user experience. Each methods have pros and cons. Tokens like JWT and OAuth 2.0 are fast, stateless, scalable. They don't need server storage and can expire for security. But tokens can be stolen, so careful management is needed. Server-side sessions, cookies, and SSO are easy to manage and let you revoke access quickly. But they can slow the system and bound scalability. Passwordless methods like biometrics, hardware keys, and SMS codes make logins safer and remove passwords. They cost more and may need special devices. OpenID Connect is good for Single Sign-On and widely used, instead it must be set up carefully to avoid fake identities. SMS codes work for two-factor authentication but can be hacked through SIM swaps. These insights can guide developers in designing secure MSA authentication flows. Based on these findings, we recommend the following:

Use a dedicated authentication service: Select Centralizing authentication for easy updates easy, helps the system scale, and consistent across security of all services.

Choose authentication methods carefully: For fast, stateless login use tokens like JWT or OAuth 2.0.(manage them carefully to avoid theft or misuse).

Consider passwordless methods where feasible: Select Biometrics, hardware keys, and SMS codes for safer and easier login (Important apps).

Plan for Single Sign-On carefully: Use OpenID Connect for easier access across services.

Address service-to-service and API authentication: Make sure that Inter service communication and API access are secure.

Tailor methods to system needs: Know your microservice setup, performance needs, and security risks to choose the best authentication methods.

Overall, microservices need flexible and strong authentication approaches to maintain both security and efficiency. By carefully combining tokens, sessions, and modern passwordless techniques, systems can scale, stay secure, and provide a smooth user experience while following today's web security best practices.

7.2 Future Research

Future research on microservice authentication should tackle new security challenges. Distributed systems keep changing and growing more complex. Researchers need to find better ways to secure them. Dynamic architectures add more risks. Simple, scalable solutions will be key. The focus should be on keeping these systems safe as they evolve. We need to explore key areas. One focus is on creating better token-based authentication. These tokens should be lightweight but strong. They must improve scalability. They should also make security between services stronger. New ideas in authorization can make service interactions safer and easier. They help reduce risks like stolen tokens or weak sessions. These improvements keep data secure while making access smoother. Simple, smart solutions can protect systems better. New passwordless login methods are improving. Options like fingerprints or face scans make logging in easier. Hardware keys are also becoming popular. These changes make things simpler for users. They also reduce the need for old-style passwords. This makes accounts safer and more convenient. Multi-factor and multi-biometric systems can make security stronger. They also keep performance high. These methods are a good way to improve safety. They use more than one check to verify identity. This makes it tough for others to break in. They also work quickly and run smoothly. Everything stays secure without slowing down. It's a good balance. As microservices grow, adding AI tools can help. These tools can spot odd behavior. They can also adjust security as needed. This helps stop threats before they cause harm. It keeps systems safer. Future research should also explore the integration of privacy-preserving techniques, such as anonymous credentials, and the maturation of federated identity frameworks like those based on Self-Sovereign Identity (SSI), to enhance user privacy within distributed authentication systems.

References

[1] K. Brown and B. Woolf, "Implementation patterns for microservice architectures," in *Proc. 23rd Conf. Pattern Lang. Programs*, 2016, pp. 1-35.

[2] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Proc. IEEE 9th Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, 2016, pp. 44-51.

[3] M.G. sde Almeida and E.D. Canedo, "Authentication and authorization in microservice architecture: A systematic literature review," *Appl. Sci.*, vol. 12, no. 6, p. 3023, 2022.

[4] A. Bánáti, E. Kail, K. Karóczkai, and M. Kozlovszky, "Authentication and authorization orchestrator for microservice-based software architectures," in *Proc. 41st Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, 2018, pp. 1180-1184.

[5] X. He and X. Yang, "Authentication and authorization of end user in microservice architecture," *J. Phys.: Conf. Ser.*, vol. 910, p. 012060, 2017.

[6] S. Daya *et al.*, *Microservices from theory to practice: creating applications in IBM Bluemix using the microservices approach*. IBM Redbooks, 2016.

[7] T. Yarygina, "Exploring microservice security," University of Oslo, Tech. Rep., 2018.

[8] M. Söylemez, B. Ticonderoga, and A. Kolukısa Tarhan, "Challenges and solution directions of microservice architectures: A systematic literature review," *Appl. Sci.*, vol. 12, no. 11, p. 5507, 2022.

[9] R.M. Munaf, J. Ahmed, F. Khakwani, and T. Rana, "Microservice architecture: Challenges and proposed conceptual design," in *Proc. Int. Conf. Commun. Technol. (ComTech)*, 2019, pp. 82-87.

[10] O. Ethelbert, F.F. Moghaddam, P. Wieder, and R. Yahyapour, "A JSON token-based authentication and access management schema for cloud SaaS applications," in *Proc. IEEE 5th Int. Conf. Future Internet Things Cloud (FiCloud)*, 2017, pp. 47-53.

[11] Y. Balaj, "Token-based vs session-based authentication: A survey," *Int. J. Comput. Sci. Trends Technol.*, vol. 5, no. 5, pp. 1-6, Sep.-Oct. 2017.

[12] Okta, "What is token-based authentication?," Okta, Inc. [Online]. Available: https://www.okta.com/identity-101/what-is-token-based-authentication/

[13] K. Shingala, "Json web token (jwt) based client authentication in message queuing telemetry transport (mqtt)," *arXiv preprint arXiv:1903.02895*, 2019.

[14] A. Bucko, K. Vishi, B. Krasniqi, and B. Rexha, "Enhancing jwt authentication and authorization in web applications based on user behavior history," *Computers*, vol. 12, no. 4, p. 78, 2023.

[15] JWT.io, "Introduction to JSON Web Tokens (JWTs)," Auth0, Inc. [Online]. Available: https://jwt.io/introduction

[16] P. Mahindraka, "Insights of JSON Web Token," *Int. J. Recent Technol. Eng.*, vol. 8, no. 5, pp. 2277-3878, Jan. 2020.

[17] Y. Balaj, "Token-based vs session-based authentication: A survey," *Int. J. Comput. Sci. Trends Technol.*, vol. 5, no. 5, pp. 1-6, Sep.-Oct. 2017.

[18] Criipto, "Session token-based authentication," Criipto ApS. [Online]. Available: https://www.criipto.com/blog/session-token-based-authentication

[19] B. Yousra *et al.*, "A novel secure and privacy-preserving model for OpenID connect based on blockchain," *IEEE Access*, vol. 11, pp. 12025-12042, 2023.

[20] O. Oleh, "Future of Identity and Access Management: The OpenID Connect Protocol," M.S. thesis, Kyiv National University of Technologies and Design, Kyiv, Ukraine, 2018.

[21] A. Alsadeh, N. Yatim, and Y. Hassouneh, "A Dynamic Federated Identity Management Using OpenID Connect," *Future Internet*, vol. 14, no. 11, p. 339, 2022.

[22] Y. Balaj, "Token-based vs session-based authentication: A survey," *Int. J. Comput. Sci. Trends Technol.*, vol. 5, no. 5, pp. 1-6, Sep.-Oct. 2017.

[23] V. Parmar, H.A. Sanghvi, R.H. Patel, and A.S. Pandya, "A comprehensive study on passwordless authentication," in *Proc. Int. Conf. Sustain. Comput. Data Commun. Syst. (ICSCDS)*, 2022, pp. 1266-1275.

[24] Logto, "Token-based authentication vs session-based authentication," Shenzhen Invariant Tech. Co., Ltd., 2023. [Online]. Available: https://blog.logto.io/token-based-authentication-vs-session-based-authentication

[25] 884m884, "Understanding session-based authentication from scratch," Medium, 2020. [Online]. Available: https://medium.com/@884m884/understanding-session-based-authentication-from-scratch-64110bcfc00f

[26] Y. Mundada, N. Feamster, and B. Krishnamurthy, "Half-baked cookies: Hardening cookie-based authentication for the modern web," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 675-685.

[27] S.K. Sood, A.K. Sarje, and K. Singh, "Inverse Cookie-based Virtual Password Authentication Protocol," *Int. J. Netw. Secur.*, vol. 13, no. 2, pp. 98-108, 2011.

[28] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 1-29, 2014.

[29] V. Radha and D.H. Reddy, "A survey on single sign-on techniques," *Procedia Technol.*, vol. 4, pp. 134-139, 2012.

[30] T. Bazaz and A. Khalique, "A review on single sign-on enabling technologies and protocols," *Int. J. Comput. Appl.*, vol. 151, no. 11, pp. 18-25, 2016.

[31] V. Parmar, H.A. Sanghvi, R.H. Patel, and A.S. Pandya, "A comprehensive study on passwordless authentication," in *Proc. Int. Conf. Sustain. Comput. Data Commun. Syst. (ICSCDS)*, 2022, pp. 1266-1275.

[32] JumpCloud, "SSO: Pros and cons," JumpCloud, Inc. [Online]. Available: https://jumpcloud.com/blog/sso-pros-cons

[33] T. Bazaz and A. Khalique, "A review on single sign-on enabling technologies and protocols," *Int. J. Comput. Appl.*, vol. 151, no. 11, pp. 18-25, 2016.

[34] V. Parmar, H.A. Sanghvi, R.H. Patel, and A.S. Pandya, "A comprehensive study on passwordless authentication," in *Proc. Int. Conf. Sustain. Comput. Data Commun. Syst. (ICSCDS)*, 2022, pp. 1266-1275.

[35] D. Bhattacharyya, R. Ranjan, F. Alisherov, and M. Choi, "Biometric authentication: A review," *Int. J. u- e-Service, Sci. Technol.*, vol. 2, no. 3, pp. 13-28, 2009.

[36] K. Dharavath, F.A. Talukdar, and R.H. Laskar, "Study on biometric authentication systems, challenges, and future trends: A review," in *Proc. IEEE Int. Conf. Comput. Intell. Comput. Res.*, 2013, pp. 1-7.

[37] C. Mulliner, R. Borgaonkar, P. Stewin, and J.P. Seifert, "SMS-Based One-Time Passwords: Attacks and Defense," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, 2013, pp. 150-159.

[38] M.A.R. Karia, D.A.B. Patankar, and P. Tawde, "SMS-based one-time password vulnerabilities and safeguarding OTP over the network," *Int. J. Eng. Res. Technol.*, vol. 3, no. 5, pp. 1339-1343, 2014.

[39] K. Sagar and V. Waghmare, "Measuring the security and reliability of authentication of social networking sites," *Procedia Comput. Sci.*, vol. 79, pp. 668-674, 2016.

[40] Auth0, "Social login," Auth0, Inc. [Online]. Available: https://auth0.com/learn/social-login

[41] J.T. Zhao, S.Y. Jing, and L.Z. Jiang, "Management of API gateway based on micro-service architecture," *J. Phys.: Conf. Ser.*, vol. 1087, p. 032032, 2018.

[42] S. Gadge and V. Kotwani, "Microservice architecture: API gateway considerations," *GlobalLogic*, White Paper, Aug. 2017.

[43] F. Montesi and J. Weber, "Circuit breakers, discovery, and API gateways in microservices," *arXiv preprint arXiv:1609.05830*, 2016.

[44] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice API Evolution in Practice: A Study on Strategies and Challenges," *J. Syst. Softw.*, vol. 215, p. 112110, 2024.

[45] A. Hannousse and S. Yahiouche, "Securing microservices and microservice architectures: A systematic mapping study," *Comput. Sci. Rev.*, vol. 41, p. 100415, 2021.

[46] Y. Chen, E. Fernandes, B. Adams, and A.E. Hassan, "On practitioners' concerns when adopting service mesh frameworks," *Empir. Softw. Eng.*, vol. 28, no. 5, p. 113, 2023.

[47] V. Stafford, "Zero trust architecture," *NIST Spec. Publ.*, vol. 800, p. 207, 2020.

[48] P. Arias-Cabarcos, C. Krupitzer, and C. Becker, "A survey on adaptive authentication," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1-30, 2019.

[49] K.A.A. Bakar and G.R. Haron, "Adaptive authentication: Issues and challenges," in *Proc. World Congr. Comput. Inf. Technol. (WCCIT)*, 2013, pp. 1-6.

[50] M.T. Hammi, P. Bellot, and A. Serhrouchni, "BCTrust: A decentralized authentication blockchain-based mechanism," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, 2018, pp. 1-6.