

Article

Authentication Challenges and Solutions in Microservice Architectures

Constantin Lucian Aldea [†] and Razvan Bocu ^{*,†}

Department of Mathematics and Computer Science, Transilvania University of Brasov, 500036 Brasov, Romania; costel.aldea@unitbv.ro

* Correspondence: razvan.bocu@unitbv.ro

[†] These authors contributed equally to this work.

Abstract

In this paper, we examine the relevant vulnerabilities and security controls for ensuring the security of applications built on microservice architectures. Zero Trust security principles are used to conceptualize and implement a secure ecosystem using Spring Boot security components and Docker infrastructure. Relevant security controls are analyzed and a proof of concept was created. The combination of security controls in the Docker environment, the deployment of these controls, and the analysis of their impact will also be the focus of this paper.

Keywords: authentication; identity and access management; microservices; cloud security

1. Introduction

Zero trust is a fundamental paradigm in information system security, which assumes that no user or device can be implicitly trusted, even if they are inside the perimeter. Zero trust applies to users, devices, and microservices. It is about ensuring secure access when an endpoint, microservice system, or container accesses data.

This paper presents results related to the authentication aspects in distributed systems and cloud environments when using microservice architectures. Considerations related to the existing authentication mechanisms and their security issues are addressed. Instructions are provided to create a sample application that uses Spring Boot security for authentication in a Docker environment to analyze the endpoint security filters.

APIs are the main channels through which applications communicate with each other, both in microservice-oriented architectures and in architectures that use external cloud services. Securing these access points is necessary to prevent unauthorized access to data and services [1].

OAuth 2.0 is an industry authorization standard that provides authentication and authorization in the communication flow between services, servers, applications, and end users.

JSON Web Tokens (JWTs) allow applications to request a token from the IAM service that grants specific permissions, called claims, without exposing credentials. Because tokens can be restricted to specific uses and have short lifetimes, this model minimizes the attack surface.

OpenID Connect is an identification layer built on top of OAuth 2.0 that allows the user's identity to be verified based on authentication performed by an authorization server and a basic user profile information to be obtained.



Academic Editors: Stefan Fischer and George Drosatos

Received: 29 September 2025

Revised: 29 October 2025

Accepted: 11 November 2025

Published: 14 November 2025

Citation: Aldea, C.L.; Bocu, R. Authentication Challenges and Solutions in Microservice Architectures. *Appl. Sci.* **2025**, *15*, 12088. <https://doi.org/10.3390/app152212088>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

The authors of paper [2] have implemented and evaluated a proof of concept of IAMaaS (IAM as a Service). IAM implemented as a cloud service can provide the user with all the benefits of Security-as-a-Service (SECaaS). In line with the cloud model, the concept was implemented as a set of virtual machines within a cloud environment.

In paper [3], integrating a service mesh with zero trust, identity and access management (IAM), encryption, and runtime protection creates a unified security architecture for cloud-native systems. The service mesh enforces encrypted and authenticated communication between services, while zero trust provides continuous verification and micro-segmentation. Furthermore, IAM ensures granular access control and multi-factor authentication (MFA), encryption secures data in transit and at rest, and runtime protection delivers real-time threat detection.

Paper [4] explores the research challenges and potential enablers for user authentication in the IoT realm, and presents a multi-factor authentication concept for IoT as an alternative to the existing single-factor solutions with limited capabilities. One can consider a similarity between Internet of Things (IoT) devices and microservices as communication actors that need to be authenticated.

Study [5] maps security vulnerabilities in microservices to the OWASP Top 10 risks, highlighting that traditional web application flaws persist and are exacerbated in distributed architectures. Critical issues such as injection attacks, broken authentication, cross-site scripting (XSS), broken access control, security misconfiguration, sensitive data exposure, and insufficiently protected APIs are prevalent in microservice environments. The paper also emphasizes that the increased attack surface and inter-service trust relationships make it harder to mitigate these risks. This necessitates robust measures, such as strong authentication, encrypted communication, defence-in-depth strategies, and continuous monitoring, to prevent exploitation across multiple services.

The security of a zero-trust network architecture often mitigates external threats while ignoring internal malicious requests within IoT or microservice clusters. Internal threats can also lead to data breaches or operational disruptions. To address this, the zero trust principle of *never trust, always verify* should be applied to all types of traffic within microservice or IoT clusters. Token-based access control and blockchain for tracing and forensics have limitations in mitigating threats. To improve protection against internal threats, the authors of article [6] propose a novel approach to intra-cluster traffic validation based on API function control. This involves setting up a verification server and embedding verification procedures at the entrance of each API function. The verification server examines incoming requests to decide whether to allow responses.

Spring Security is one such module that provides comprehensive security features for Spring-based applications. It provides a wide range of authentication and authorization options, including support for various authentication mechanisms such as LDAP, OAuth, and OpenID [7]. Spring Security also includes features such as user account management, password encryption, and access control to protect web applications against various types of attacks. CORS (cross-origin resource sharing) is a security measure implemented by browsers that restricts access to server resources by clients from different origins (i.e., domains, protocols, or ports).

JSON Web Token (JWT) is an open standard (RFC 7519) [8] that introduces a compact and secure way to transfer information between communication partners in JSON format. A JWT token consists of three parts: a header, a payload and a signature (header, body, signature).

In paper [9], the JWT penetration testing was performed using the Cross-Site Request Forgery (CSRF) technique, revealing that an attacker could control a victim's account and compromise resources. While JWT is a common standard for user authentication and authorization, storing tokens in cookies is a vulnerability. To mitigate this risk, HttpOnly

cookies can be used. HttpOnly cookies are a special type of cookie that are sent to the server in HTTP requests and are inaccessible to JavaScript running in the browser, thereby increasing security. Additionally, the contribution that was reported in article [10] highlights the relevance of authentication processes in microservice-based architectures.

The contribution presented in book [11] concerns a practical methodology for intelligent college construction and development processes. The solution involves the design and implementation of proper intrusion detection systems and microservice architecture models. The authors claim that the resulting microservice-based authentication system significantly improves the security of the students' private data while preserving their pleasant user experience.

The solution in [12] concerns a secure service discovery architecture, which includes security mechanisms at each phase of the discovery process. This solution encompasses encryption models and authentication and authorization subsystems to protect microservice architectures in IoT environments. Furthermore, a mathematical model is presented to design and implement the necessary security mechanisms. This approach may be sufficient to mediate the reliable authentication-based microservice architectures relative to the IoT ecosystem. Additionally, it is relevant to note that the authors of paper [13] reported a systematic scientific survey on relevant security aspects in microservice architectures.

The OpenRAN initiative specifies an open Radio Access Network (RAN) and provides operators with fine-grained configuration relative to the radio stack. Thus, O-RAN specifies new components for the 5G ecosystem, for example, the near-real-time RAN Intelligent Controller (near-RT RIC), and the related extensible Applications (xApps). The specification of these components significantly develops the 5G threat surface. Moreover, the shift from proprietary hardware to virtual environments using Network Functions Virtualization (NFV) creates further attack vectors. Consequently, the authors of paper [14] reported the xApp repository function (XRF) framework, which is used to specify and implement scalable authentication, authorization, and discovery of processes of xApps. The hardening of XRF microservices is achieved using the Intel Software Guard Extensions (SGX, version 2.25.100.3). Although the isolation of XRF microservices determines adequate logical authentication models, the induced overhead makes this approach computationally inefficient in certain real-world use case scenarios.

Paper [15] outlines the distinctive security and identity issues posed by agentic AI within zero-trust microservices, including ephemeral identities, fluctuating workloads, and machine-to-machine interactions. It also presents a decentralized architecture for authentication and authorization.

Consequently, this paper presents an efficient microservice-based architecture, which addresses the identified shortcomings of the existing solutions regarding the authentication and data security models in microservice-based architectures. In the context of Spring Security, JWT tokens are often used for authentication and authorization in web applications. Typically, when a user logs in, a JWT token is generated and sent to the client. The client then includes the JWT token in subsequent requests to the server, which verifies the token signature and extracts the user information from the token body to authenticate and authorize the request.

The contributions proposed in this paper are the following:

- Proof of concept for a microservice architecture using an IAM service.
- Specification of security controls, aligned with zero trust principles that are suitable for microservice ecosystems, and the corresponding IAM services (e.g., Spring Boot and Docker).
- The zero trust principles were strengthened in relation to a list of improper design decisions, which were classified according to their security impact.

2. Security Landscape

2.1. Identity and Access Management (IAM)

The role of identity and access management (IAM) is to ensure that an organization's users have access to the resources and information they need to do their jobs, facilitating the implementation of needs-based security and the zero trust model while reducing the risk of data exposure. IAM is about managing digital identities and access to resources in an efficient, secure, and compliant manner.

Identity access management (IAM) provides a way to control what users can access, ensuring that verified entities have the right level of access to the system's resources. It involves two steps: identity management and access management. Identity management ensures that the person attempting to access the system matches the identity they claim to hold. This is done via logging in to the system and is, nowadays, supplemented with multi-factor authentication (MFA) for added security. Following this identity check, access management identifies the right level of access the user holds and ensures that all resources accessible to that user belong to that access level, in a process called authorization. In summary, IAM systems ensure the correctness of authentication and authorization processes.

Among the main security controls and countermeasures of IAM are the following:

Access control (ensures that only authorized users have access to resources and data based on authentication and access authorization by implementing access control mechanisms).

Data protection (use of countermeasures to prevent unauthorized access, theft, or compromise of information. This includes data encryption, identity and access management, and the enforcement of security policies).

Regulatory compliance (compliance with legal requirements and support in the regulatory process).

Monitoring and auditing (auditing of relevant security events and integration with intrusion detection and prevention systems to recognize and prevent malicious activities).

The architecture and structure of an IAM system is complex and varies depending on the specific needs of an entity [2]. An IAM system consists of interconnected components that work together to manage user identities and access to resources. These components include the following:

- Directory services—Store and manage identities, assign rights based on policies (e.g., Active Directory).
- Authentication—Confirm identity (passwords, tokens, multi-factor authentication, biometrics).
- Authorization—Grant access permissions to authenticated users using role-based rules.
- Single Sign-On (SSO)—Access to applications based on a single authentication using an access token.
- Identity Management—Identity lifecycle management, including access creation and revocation.
- Access and monitoring—User access management and monitoring with detection and reporting of suspicious activity.

2.2. IAM Standards

IAM frameworks unify authentication and authorization system components. For simplicity, an IAM architecture combines identity (user management and authentication) and access (authorization) management models (see Figure 1) into an ecosystem that fits the needs of the business/enterprise employing it [16].

To ensure consistency and interoperability, IAM solutions often rely on established standards and protocols. Protocols used at large for authentication include MFA, OpenID, SAML, and OAuth. These digital ways of identification reduce the risk of a malicious party

gaining access to the internal network of an enterprise and allow, to some degree, to be used interchangeably.

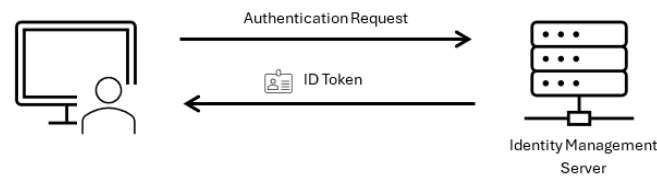


Figure 1. IAM authentication flow.

This paper introduces a conceptual framework to enhance understanding of identity and access management (IAM) through the integration of identity management (IdM) and access control mechanisms (ACMs).

2.3. Zero Trust Architectures

With the high usage of cloud technologies and the shift towards working remotely, the network location is no longer the primary focus of security measures [17]. As such, Zero Trust Architectures (ZTAs) focus on protecting resources rather than network segments. This represents a transition from traditional perimeter-based security architecture, which relies on firewalls to separate internal and external networks and deploys intrusion detection systems at the network boundary. ZTAs work under the assumption that a malicious party is already present within the network, or simply put, that the company's internal network is just as trustworthy as the external one. This means that there is no notion of implicit trust, and that each access request needs to be authenticated and authorized, thereby decreasing the risk authenticated malicious objects pose in PSAs [18]. To this end, implementing a ZTA follows two major goals: *prevention of unauthorized access* and *granular access control*.

2.3.1. Authentication

Authentication mechanisms are based on knowledge, inherence, and/or possession. Traditional password-protected mechanisms are prime examples of authentication based on knowledge. However, there are several vulnerabilities associated with them: users are known to choose very simple passwords [19] and with the number of accounts a person usually has, choosing strong and unique passwords for each of them is a difficult task [20]. Inherence factors rely on physical and behavioral patterns of the user, which, while more difficult to guess compared to knowledge-based ones, are impossible to reset once compromised, i.e., if the fingerprint of the user is recreated by a malicious party, any present and future use of the fingerprint for identification purposes is compromised. Hardware tokens are representative for possession-based authentication methods [21]. Such methods require the user to own a specific piece of hardware, which generates a dynamic one time password (OTP) required for authentication. However, this is considered burdensome both for the user and the service provider as it introduces the need to always carry that device and incurs additional costs. As an alternative, the use of mobile phones as dedicated hardware devices has been explored; however, this only serves to shift the vulnerability, as using the corresponding application on the phone also requires the user to have proven their identity [22].

More recent advances in authentication mechanisms explore the use of digital memory; however, this approach presents some usability issues as a questionnaire approach is time consuming (9 s per question) and introduces power drainage issues. This approach also raises privacy concerns, as identifying a correct answer requires in-depth knowledge of the user activity. More authentication methods proposed and implemented include facial recognition using 3D camera technologies, proven vulnerable to 3D-printed heads of a

person; breathing gestures; ear canal shape; and EEG signals considered infeasible for usual usage scenarios. Due to the vulnerabilities or limited applicability of the mentioned methods, focus has shifted towards multi-factor authentication (MFA). The core idea behind MFA is that if one of the authentication methods is compromised, the other supplementary authentication methods required, often of a different type than the main one (knowledge, inference, or possession), ensure that the risk posed is decreased. It is to be noted that MFA methods introduce a sense of reliability on an external tool to be able to connect to a service, or more simply put, if there is no access to the MFA device generating the one time code (OTC), then the service becomes inaccessible [23].

2.3.2. Authorization

Access control models are used to decide what resources an authenticated user is allowed to access. The standard and four most common authorization models are mandatory access control (MAC), discretionary access control (DAC), role-based access control (RBAC), and privileged access management (PAC). Each of these models presents advantages and disadvantages, allowing a service provider to choose which one fits their requirements best.

MAC assigns labels to resources and entities (users and devices), which represent the compartment and the classification/clearance. This system is used to describe a resource's sensitivity and the compartment of users that could have access to it based on their individual clearance. This system is easy to enforce while providing the granular access control required by ZTA; however, it achieves compartment security by restricting the communication of resources. Furthermore, it requires a dedicated structure for managing security labels, which could be considered an additional organizational burden. On the other hand, DAC decentralizes the resource management, with the access list of a resource being set by the owner of that particular resource. While it brings various advantages with its conceptual simplicity, most notably its responsiveness to business needs, the lack of centralized oversight can introduce security vulnerabilities as users are given control over access policies. This is particularly problematic as it could lead to inconsistencies. For example, to access users' privileges, every existing access control list must be searched, and changes to these lists follow the same process.

RBAC is more complex: it requires defining different roles in the organization and determining what resources individual roles have access to. It is more flexible than the MAC as it allows the creation of hierarchies by assigning individual users multiple roles and not restricting access based strictly on the compartment. RBAC policies can be set consistently within an organization, which allows for a centralized management of roles. Additionally, if an entity's role within the organization changes, it is only required that his role is adjusted accordingly, i.e., removing the old role and assigning the new one. However, complex deployment scenarios require the creation and management of very specific roles which can prove challenging and has introduced the need for role engineering. However, RBAC does not remove the risk of over-privileged users; this is something that PAM attempts to mitigate. PAM functions on strict 'least privilege' access principles, meaning that entities have limited access to resources, access being restricted to the strictly needed resources. While this reduces the threat surface, the implementation of PAM is meeting some resistance due to its high complexity and implementation costs [24].

2.4. Security Frameworks, Protocols and Standards

Spring Security is closely integrated with the Spring ecosystem, including Spring Boot and Spring MVC. It offers comprehensive support for authentication and authorization, and features such as CSRF protection, session management, and method-level security. It also supports OAuth2, SAML and LDAP (Table 1).

Table 1. Security frameworks, protocols, and standards.

Name	Framework/ Protocol/Standard	Use For	Key Features	Integration
Spring Security	Framework	Spring-based enterprise apps	Full-stack security, OAuth2, OIDC, SAML, LDAP, JWT support	Deep (Spring)
Apache Shiro	Framework	Lightweight, non-Spring, or simple apps	Auth, authz, crypto, session mgmt, basic JWT	Flexible
Keycloak	Framework (IAM)	Centralized IAM, SSO, federation	SSO, OAuth2, OIDC, SAML, admin UI, user mgmt, JWT	Any (external provider)
Java EE Security	Framework	Java EE apps	Declarative security, container-managed	Java EE
JWT	Standard	Stateless APIs, microservices	Token-based stateless auth, easy to use, widely supported	Any (standard)
OAuth2	Protocol	Delegated authorization, APIs, SSO	Authorization protocol, scopes, tokens, supports SSO, often used with OIDC	Any (standard)
OpenID Connect	Protocol/Standard	SSO, federated identity, user profile	Authentication protocol, built on OAuth2, ID tokens, user info endpoint	Any (standard)

Apache Shiro can be used with or without Spring. It integrates well, but is not tied to any framework. It offers features such as authentication, authorization, cryptography, and session management. It lacks out-of-the-box support for OAuth2 and SAML.

Keycloak is a standalone identity and access management (IAM) solution. It contains features for centralized authentication and authorization, single sign-on (SSO), social login, user federation, role-based access control, an admin UI, OAuth2, OpenID Connect, and Security Assertion Markup Language (SAML).

JWT is a token format and standard, rather than a complete security framework. It is commonly used alongside frameworks such as Spring Security, Shiro, and Keycloak for stateless authentication.

Similarly, OAuth2 is an authorization protocol, not a framework. These frameworks implement OAuth2 to provide delegated access and single sign-on (SSO).

OpenID Connect (OIDC) is an authentication protocol that builds on the authorization capabilities of OAuth 2.0. It enables clients, such as web and mobile applications, to verify users' identities based on authentication performed by an authorization server and obtain basic profile information.

Spring Security should be considered when an existing project is based on Spring and requires advanced, customizable security features. For projects that are simpler, lightweight, or not based on Spring, Apache Shiro is recommended. Keycloak is suitable for scenarios requiring centralized identity and access management (IAM), single sign-on (SSO), or when user management needs to be offloaded. For environments centered on Java EE or JBoss, the Java EE Security API, JAAS, or PicketLink should be utilized.

Cloud computing is a dynamic, distributed system that offers scalable, cost-effective and flexible services. This has led to its rapid adoption by organizations. However, despite its advantages, cloud environments face significant security and privacy challenges, primarily due to multi-tenancy and third-party infrastructure. Robust identity and access management (IAM) is, therefore, essential. The paper [25] surveys the

authentication and access control techniques used by cloud providers and their users, providing comparative insights.

This paper [26] examines the operation of identity and access management (IAM) in multinational information-sharing environments such as the US Department of Defense (DoD). It emphasizes the need to strike a balance between facilitating secure information exchange and protecting against unauthorized access and cyber threats. The study explores the role of IAM in enabling secure collaboration and supporting zero trust frameworks, and standardizing authentication and authorization across diverse systems and entities, including individuals and non-personal entities.

The Open Web Application Security Project (OWASP) identifies what needs to be secured. The zero trust model, conversely, defines how this should be achieved in dynamic, distributed environments.

Figure 2 maps the OWASP Top 10, which identifies critical web application security risks such as injection attacks, broken authentication, exposure of sensitive data, and under-protected APIs. The zero trust principles of verify explicitly, least privilege, assume breach, micro-segmentation, and encrypt in transit and at rest provide a proactive framework for countering these risks.

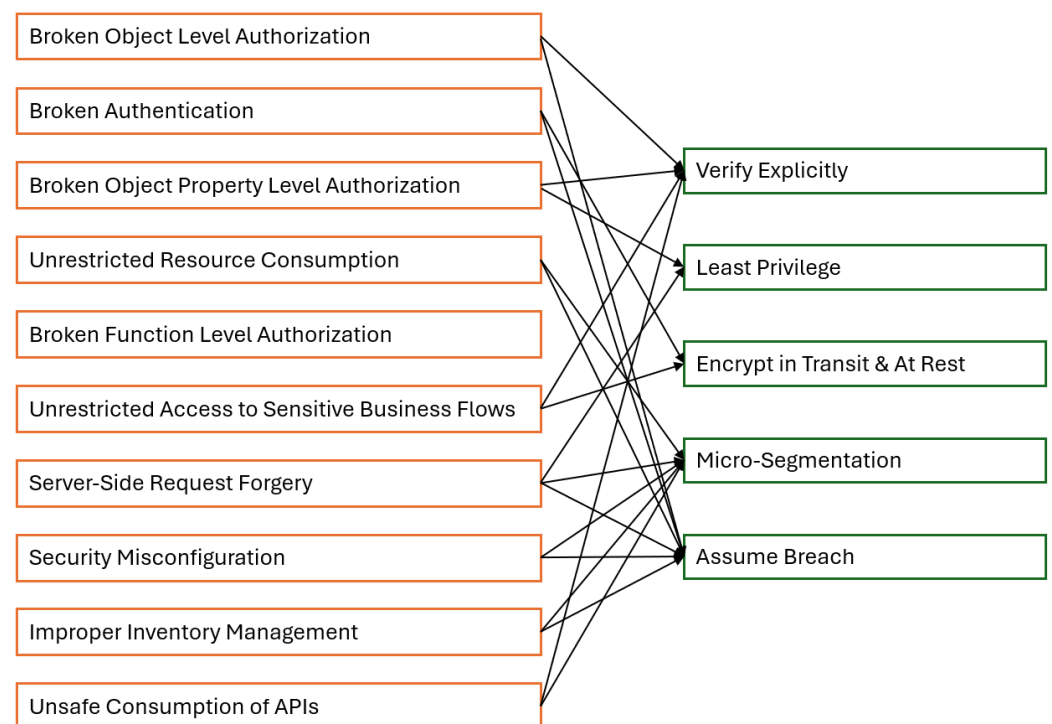


Figure 2. OWASP vs. zero trust in microservice security.

2.5. Multi-Factor Authentication

Multi-factor authentication (MFA) is an access control security mechanism that uses at least two different authentication methods, such as a password and a dynamically generated token or biometrics. MFA is a security strategy that increases the level of protection by introducing additional layers of authentication, even in situations where one of the authentication keys has been compromised [27].

The banking and finance industry was one of the first to adopt MFA, using hardware tokens that generate unique access codes. Social media platforms such as Facebook, Twitter and LinkedIn typically use authentication apps or SMS authentication. Gmail and Outlook offer MFA options to protect user accounts by generating tokens in authenticator apps.

Many security standards and regulations, such as GDPR (General Data Protection Regulation) and HIPAA (Health Insurance Portability and Accountability Act), require the use of MFA to protect sensitive data. Implementing MFA helps organizations comply with these regulations and avoid fines or penalties.

Although MFA provides a high level of security, it is subject to attacks such as phishing (illegitimate messages that allow attackers to obtain the authentication token in addition to the password), SIM swapping (replacing the SIM card with a copy), man-in-the-middle (interposing between communicating actors by intercepting or even forging exchanged messages), social engineering (manipulating users to obtain authentication information or session tokens), and use of backup codes (bypassing MFA authentication). Some of the attacks can be avoided by introducing an expiration time for passwords or secrets exchanged by services or users (Time-Based One-Time Password).

3. Case Study for IAM Framework Using Docker

Using a case study, the security controls of the architecture are analyzed for a practical IAM framework deployed using Docker containers, enabling modular, scalable, and isolated management of authentication and authorization services within a microservice environment (Figure 3).

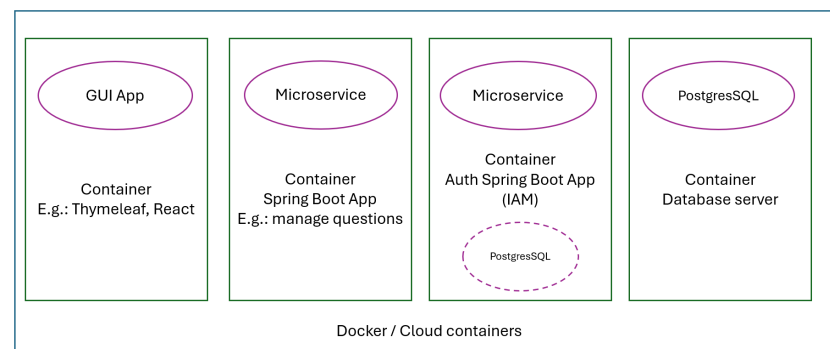


Figure 3. Microservice authentication architecture.

IAM refers to the frameworks, policies, and technologies that ensure the right individuals (or entities) have the appropriate level of access to technological resources at the right time and for the right reason. The Figure 4 depicts the IAM architectural stack.

By using Spring Boot, one can design and implement Java-based microservices. This proof of concept analyzes an architecture as shown in Figure 3.

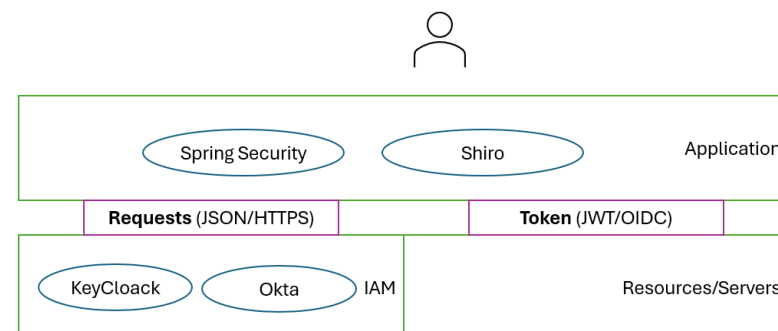


Figure 4. IAM architecture.

Using the *Spring Boot Initializr* [28], a microservice project is configured by filling in the fields under Project Metadata and adding the necessary dependencies. In the case of this project, Apache Maven was chosen because it is a build and management system suitable for the complexity level of this application. The dependencies added include

Spring Web, the driver for the database used (PostgreSQL), Spring Data JPA to enable work with relational databases, and Lombok to reduce boilerplate code (it makes the code easier to read and reduces development time).

3.1. Business Scenario

The custom IAM solution can be used in financial, health, energy, and other business domains. One example of a business process involves IAM, security frameworks, and adaptive policies working together to provide flexible, proactive, and secure protection in a microservice-based business environment (Figure 4).

User Login—A customer logs in to the web portal. Their authentication request is then redirected to IAM, where their credentials are verified, and an OIDC-compliant JWT token is issued.

Accessing Microservices—The user requests their account summary. The web app then forwards the JSON Web Token (JWT) to the account microservice, which uses Spring Security to validate the token and authorize access.

Adaptive IAM Policy—The ecosystem has implemented adaptive IAM policies. If the system detects unusual behavior, such as a login from a new country or rapid, high-value transactions, it will trigger additional verification steps, such as multi-factor authentication (MFA) or temporary access restrictions.

Machine Learning-Based Detection—The platform uses machine learning models to analyze user behavior across microservices. If an anomaly is detected (e.g., a fraudulent pattern), the IAM system can automatically restrict access, notify administrators, and request further verification from the user.

Audit and Compliance—For compliance and auditing purposes, all access and policy changes are logged, thereby ensuring that the system meets all regulatory requirements.

3.2. Improper Design Decisions

The distributed nature of microservices increases security risks due to multiple endpoints and inter-service communication.

The most widely recommended solutions are the existing standards and patterns, such as OAuth, TLS, and JSON Web Tokens (JWTs), and API gateways [29].

Refactoring is required to mitigate security risks arising from improper design decisions in microservice-based applications.

Table 2 shows the correspondence between improper design decisions and their security countermeasures [29], and the zero trust principle that encompasses the countermeasures when applied.

Table 2. Security improper design decision → refactorings → zero trust alignment for microservices.

Improper Design Decision	Refactoring	Zero Trust Principle	Affected Property
Missing access control	Use OAuth 2.0	Verify explicitly (token validation per call), Least privilege (scopes)	Confidentiality
Exposed microservices	Add API Gateway	Assume breach (single choke point), Micro-segmentation & policy enforcement, Verify explicitly	Confidentiality
Excessive privileges	Apply Least Privilege Principle	Least privilege (minimized permissions), Assume breach (reduced blast radius)	Confidentiality, Integrity

Table 2. Cont.

Improper Design Decision	Refactoring	Zero Trust Principle	Affected Property
Custom cryptography	Use Established Encryption Libraries	Assume breach (reduce crypto flaws), Encrypt in transit/at rest (standards-based crypto)	Confidentiality, Integrity, Authenticity
Unencrypted sensitive data	Encrypt Data at Rest	Encrypt at rest, Assume breach (data unreadable if accessed)	Confidentiality, Integrity, Authenticity
Hardcoded secrets	Encrypt Secrets at Rest; avoid storing in code/env	Verify explicitly (workload identity to fetch secrets), Least privilege (scoped access), Assume breach (no plaintext secrets)	Confidentiality, Integrity, Authenticity
Insecure inter-service communication	Use Mutual TLS (mTLS)	Verify explicitly (service identity), Encrypt in transit, Micro-segmentation (authenticated east-west traffic)	Confidentiality, Integrity, Authenticity
Unauth. traffic	Use mTLS + OpenID Connect (OIDC)	Verify explicitly (user & service identity per request), Encrypt in transit	Authenticity
Multiple authentication points	SSO via API Gateway + OIDC	Verify explicitly (centralized auth), Assume breach (consistent controls), Least privilege (token scopes/claims)	Authenticity
Centralized-only authorization	Decentralized Authorization (e.g., JWT at each service)	Verify explicitly (enforce at every hop), Micro-segmentation (policy close to resource), Least privilege (fine-grained checks)	Authenticity

3.3. Docker

Docker provides portability through application containerization, speed through rapid boot time, scalability through efficient orchestration, and consistency across environments to enable continuous software delivery.

The docker-compose.yml and Docker compose files are created with the necessary settings to run the application and database in separate containers. The environment variables are defined, and then the application.properties file is updated with the settings for the database connection, the PostgreSQL driver, and the dialect used by Hibernate. The log level is also set to debug to make it easier to find the source of any errors or failed tests.

Unlike MFA (multi-factor authentication), which relies primarily on user-provided credentials and additional verification factors for access control, Docker enables containerized environments where authentication layers are enforced at the infrastructure level. By integrating IAM as a service, API access controls can be efficiently managed through role-based access control (RBAC), ensuring consistent and scalable security policies and reducing dependence on human-based authentication. Additionally, the use of JSON Web Tokens (JWTs) in a Dockerized microservice architecture contributes to a security framework that supports stateless authentication, enabling efficient and scalable access control, while cross-origin resource sharing (CORS) policies ensure that only authorized requests

from trusted origins are processed. When combined with the zero-trust model, which assumes no implicit trust even within internal networks and enforces continuous verification, Docker may enhance security for distributed applications, including IoT environments where traditional MFA presents limitations.

One of the notable benefits of using Docker for security is its capability to implement single sign-on (SSO) while maintaining strict access controls through IAM policies. Unlike traditional MFA, which often requires multiple verification steps for each session, SSO within a containerized environment is enforced through identity providers that validate access at the API gateway level [30]. Spring Security, commonly used for securing Java-based applications, can be integrated with Dockerized applications to streamline authentication and authorization. Furthermore, Docker enhances security by enforcing policy-driven authentication through container orchestration tools such as Kubernetes, which integrates IAM at the cluster level to prevent unauthorized access. This may reduce the need for repetitive MFA steps and contribute to a more centralized and manageable security model. By incorporating Docker into IAM services, organizations can centralize identity control and enforce access policies consistently across various platforms, reducing security risks associated with lost authentication devices, such as mobile phones, and ensuring continuous protection through infrastructure-level identity verification.

Requiring a secondary authentication device is one of the methods that enhance security in MFA; however, this additional requirement can present challenges for users, such as delays in accessing their devices or difficulties when the secondary device is unavailable. This can lead to frustration, particularly in situations where quick authentication is needed but external verification steps disrupt the authentication process. A user needing access only to their laptop for work may still have to be forced to use their personal phone for authentication. This added step can slow access and reduce efficiency, especially in trusted environments. Docker addresses these security concerns by implementing an identification layer at the application level through containerized identity management. This is achieved by integrating IAM solutions directly into containerized applications, ensuring that authentication and authorization mechanisms are handled at the service level rather than depending primarily on user credentials. Through API gateways and encrypted service-to-service authentication, Docker enforces identity verification dynamically, allowing microservices to authenticate each other using secure tokens instead of static credentials. This method minimizes human intervention, reduces attack surfaces associated with compromised user accounts, and strengthens application-layer security. Instead, security policies can be dynamically enforced through encrypted service-to-service authentication using API gateways and container-specific IAM settings. This approach enhances security by ensuring strict authentication and authorization mechanisms without excessive reliance on user credentials. Environment variables are securely defined and managed within containerized environments, ensuring that sensitive credentials and configuration details remain protected from unauthorized access and potential security breaches. As a result, Docker contributes to a more resilient security framework in containerized applications by strengthening authentication and authorization mechanisms at the infrastructure level.

Dockerfile

A Dockerfile configuration has been defined to facilitate the building and execution of a Spring Boot application utilizing newest Java and PostgreSQL versions. The build process is optimized by omitting test execution, and environment variables are used to enable flexible configuration across different deployment environments. These variables also

ensure the PostgreSQL database is properly available and configured, which is essential for the application's functionality.

Key sections in defining a Dockerfile include specifying the base image (e.g., OpenJDK with Java 21), setting the working directory, copying project files, installing Maven, compiling the application, deploying the JAR, exposing ports (e.g., 8080 for Spring Boot, 5433 for PostgreSQL), configuring environment variables, and declaring the entry point to run the application.

Sample Docker compose file: The Docker compose file can be used to define a microservice-oriented (or other) architecture for Figure 3. Environment variables are used to initialize services without modifying the microservice source code. The use of volumes within the container definition enables data persistence, allowing applications to persistently store and retrieve data. In addition, using the *depends_on* option ensures that the database is available before the application starts, minimizing connectivity issues.

To ensure the security of Docker and its containers [31], it is necessary to implement the following security policy rules: continuous updates, avoidance of docker.sock exposure, use of unprivileged users to prevent privilege escalation attacks, maximum limitation of a container's functionality by applying need-to-know and zero trust principles, blocking privilege escalation, configuration of read-only volumes, and use of Docker secrets.

3.4. Authentication and Authorization

To create the authentication microservice, the first step is to add the security dependencies to the pom.xml file and then create the internal architecture based on the Spring Boot stack (Figure 5).

Next, the user entity is created, extending the user details object provided by Spring Security and using the override annotation to override the default properties of this object. The necessary annotations are also added to this entity to be able to use getters and setters and to define the name of the table in the database.

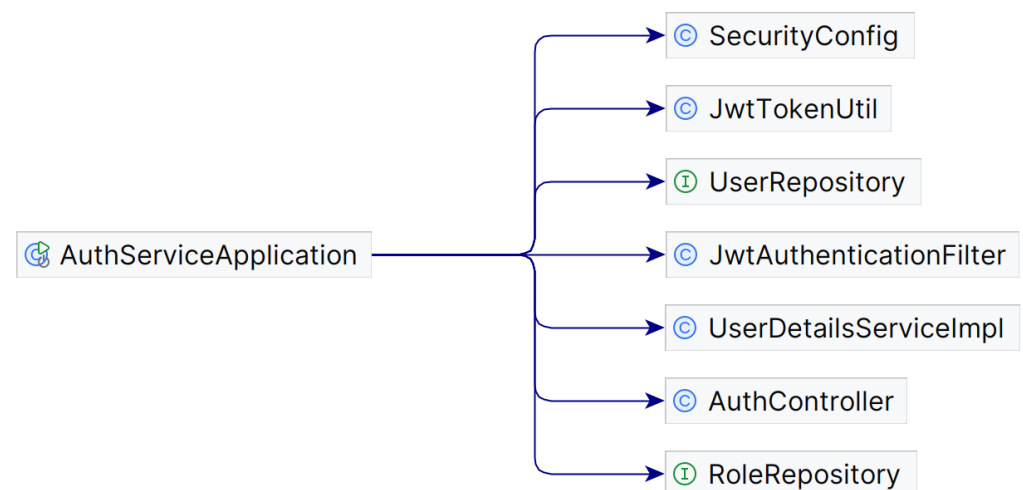


Figure 5. Spring context dependencies for authentication microservice.

The name of the table with users avoids conflicts with the default PostgreSQL table. Another important detail is the definition of a primary key id using the *@Id* and *@GeneratedValue* annotations.

Also in this phase, the types package is added and a new enum called Role is created, which is used in the user entity together with the *@Enumerated* (*EnumType.STRING*) annotation.

Then the UserRepository interface is declared which extends JpaRepository.

A security package is created to manage the security options.

The access token is included in each request-response pair. The refresh token is securely stored in a database. The flow continues if either the access token or the refresh token is valid. If the access token is no longer valid, but the refresh token is, it is used to generate a new access token.

If both tokens are invalid, or if a CSRF token cannot be found, the authentication process ends and a standard response with an http status code of 403 is sent to the client.

Filter packet: A custom filter that extends `OncePerRequestFilter` can be implemented by overriding the `doFilterInternal()` method. The `JwtAuthenticationFilter`, annotated with `@Component` and `@RequiredArgsConstructor`, is registered as a Spring bean. The filter checks for an authorization header starting with *Bearer*, extracts the token, and retrieves the user's email via the `JwtService` for further authentication processing.

Service packet: The handling of token operations is implemented using a dedicated `JwtService` class, which is annotated with `@Service`. It extracts and validates JWTs using a signing key to ensure their integrity and authenticity. The token is generated using the user's email address and a creation timestamp, and has a 24 h expiry period. Validation checks for expiry. The signing key is derived from a securely generated AES-256-CBC secret. In the authentication filter, if the user is not authenticated, their details are retrieved using the email address embedded in the token.

Config Packet: Contains the filter and service packets.

The application configuration class is created and is annotated with a configuration and requires a constructor.

A user details service is implemented, which handles the user details on the basis of an e-mail or generates a user not found error message.

Incoming requests to the Spring-based microservice are first processed by a security filter chain. This chain configures CSRF protection and CORS policies, and access rules for exposed routes.

If the access and refresh tokens are invalid, but also if a CSRF token is not found, the authentication process is terminated, and a standard HTTP status 403 response is sent to the client.

When a request passes through these steps, the credentials are retrieved from it and checked to see if they are in the server database. This process involves obtaining a provider for the authentication manager, which uses an authentication token, an object created using the credentials, and a password encoder that we specify to determine whether or not a user exists in the database. This token is also associated with the security context, which is a central class for storing authentication information about the current user.

When a user attempts to access a protected resource, Spring Security generates a unique CSRF token and stores it in the user's session. This token is then included in all subsequent requests to the server as a header. When a request is received, Spring Security compares the token in the header with the one stored in the user's session. If they match, the request can proceed. If they do not match, the request is rejected, thus preventing CSRF attacks.

3.5. Zero Trust Principles Mapped to Spring Boot and Docker Configurations

The zero trust principles are reflected in the Spring Boot and Docker functionalities, Table 3, offering the security controls needed to secure microservice architectures and enable secure identity and access management (IAM) implementations.

The zero trust principles studied and analyzed in the current paper (see also Table 2) and their corresponding security controls in Spring Boot and Docker are presented in the following sections.

Table 3. Zero trust principles applied to Spring Boot and Docker configurations.

Zero Trust Principle	Spring Boot Configuration	Docker Configuration
Verify Explicitly	Enforce Security Filter Chain; JWT validation; CSRF & CORS policies; OIDC integration	API Gateway auth; Docker secrets; Read-only mounts; Health checks
Least Privilege	RBAC/ABAC; Scoped short-lived JWTs; Data scoping; Deny-by-default rules	Non-root containers; Drop capabilities; Network segmentation; Minimal ports
Assume Breach	Token expiry/refresh; Audit logging; Step-up auth on anomalies; Fail-closed defaults	Immutable images; Continuous image scanning; Key rotation; Egress allow-lists
Micro-Segmentation	API Gateway policies; mTLS for service-to-service; Separate auth/data flows	Dedicated Docker networks; Sidecar proxies; Network policies
Encrypt in Transit & At Rest	HTTPS/TLS; Key rotation; DB encryption; Strong password hashing	TLS for inter-container traffic; Docker secrets; Encrypted volumes

3.5.1. Verify Explicitly

The *verify explicitly* principle ensures strong and ongoing authentication and authorization for each request for users, services, and devices. The Table 4 provides a list of related security controls.

Table 4. Verify explicitly security controls.

Spring Boot	Docker Infrastructure Configurations
○ Apply the security filter chain to every request. ○ Use a dedicated authentication filter to validate JWTs (issuer, audience, signature and clock skew). ○ Where applicable, enable CSRF protection and implement strict CORS policies for origins, methods and headers. ○ Integrate OIDC for identity management, per-request authorization and centralized audit logs.	○ Route all traffic via the API Gateway/Ingress for the purposes of authentication, authorization and throttling. ○ Use Docker secrets for keys and tokens. ○ Use read-only mounts and health checks to ensure that requests are not served until the authentication services are ready.

3.5.2. Least Privilege

The *least privilege* principle states that access should be granted only to the minimum necessary, and that short-lived, scoped tokens should be preferred. The Table 5 provides a list of corresponding security controls.

3.5.3. Assume Breach

The principle *assume breach* rule is that systems are designed so that compromising one component does not result in a wider breach. This involves protecting keys and data, and adding choke points. The related security controls are given in Table 6.

Table 5. Least privilege security controls.

Spring Boot	Docker Infrastructure Configurations
○ RBAC/ABAC (method and endpoint security, and fine-grained claims). ○ Scoped, short-lived JSON Web Tokens (JWTs) with minimal claims and narrow audiences. ○ Data scoping at the repository and service layers, and principal-based rate limits and deny-by-default.	○ Non-root users, capability drops and limit processes actions (seccomp / AppArmor). ○ Minimal ports and container permissions. ○ Per-service database users. ○ Network segmentation to required dependencies only.

Table 6. Assume breach security controls.

Spring Boot	Docker Infrastructure Configurations
○ Token expiry and refresh, and immediate revocation. ○ Structured audit trails with integrity protections. ○ Step-up authentication in the event of anomalies, with fail-closed defaults and minimal error leakage.	○ Read-only root file system and immutable images. ○ Continuous scanning of images and Software Bill of Materials (SBOM), and rotation of keys and secrets. ○ Egress allow-lists. ○ A kill switch and auto-containment are available for suspected compromise.

3.5.4. Micro-Segmentation

The *micro-segmentation and policy enforcement* principle is used to enforce policies. These policies are applied to each service or segment. Examples include the API gateway, sidecars, and service mesh. Relevant security controls are listed in Table 7.

Table 7. Micro-segmentation security and policy enforcement security controls.

Spring Boot	Docker Infrastructure Configurations
○ Policy enforcement at the API gateway (routes, scopes, and client identity). ○ Mutual TLS (mTLS) between services and service identities. ○ Authentication flows are separated from data flows across bounded contexts.	○ Use dedicated Docker networks for each context. ○ Where applicable, use a sidecar or proxy for policy and telemetry. ○ Implement network policies and allow lists for paths.

3.5.5. Encrypt in Transit and at Rest

The *encrypt in transit and at rest* principle states that data must be encrypted both in transit and at rest. Use TLS/mTLS for traffic, and standards-based encryption for stored data and secrets. Relevant security controls are listed in Table 8.

3.6. Performance Benchmarking

Performance is directly influenced by the number and quality of the security controls implemented. Security layers introduce additional processing, such as token validation and user context loading, resulting in higher latency. Additionally, enabling and extending the security measurement increases CPU usage due to cryptographic operations, session management, and policy enforcement. Furthermore, security frameworks consume

more memory to store user sessions, tokens, and access control data. The performance measurements were conducted on a local machine with an Intel Core i7 processor and 32 GB of RAM.

Table 8. Encrypt in transit and at rest security controls.

Spring Boot	Docker Infrastructure Configurations
○ Use HTTPS/TLS with HTTP Strict Transport Security (HSTS) and strong ciphers. ○ Rotate and protect TLS and signing keys. ○ Encrypt sensitive fields at database/application level and use strong password hashing.	○ Use TLS for communication between containers. ○ Secrets can be managed either via Docker Secrets or via an external vault. ○ We also offer encrypted volumes, backing storage, and host disks.

Although security layers enhance protection, they also introduce measurable overhead in terms of latency, CPU usage, and memory usage. These trade-offs must be taken into account when designing scalable microservices, particularly in high-performance environments.

Deviating from these cybersecurity measures can create vulnerabilities in the security architecture that malicious attackers may exploit. Such situations can have serious consequences for the company and the individuals involved.

The auth-performance-test container from Figure 6 was implemented and ran performance tests against the authentication service. It is defined in Docker and depends on the following other containers: the main authentication microservice being tested (auth-service), sample-service, database-service (additional services involved in test scenarios for service comparison and authenticated calls), and the database container that provides persistent storage for the services.

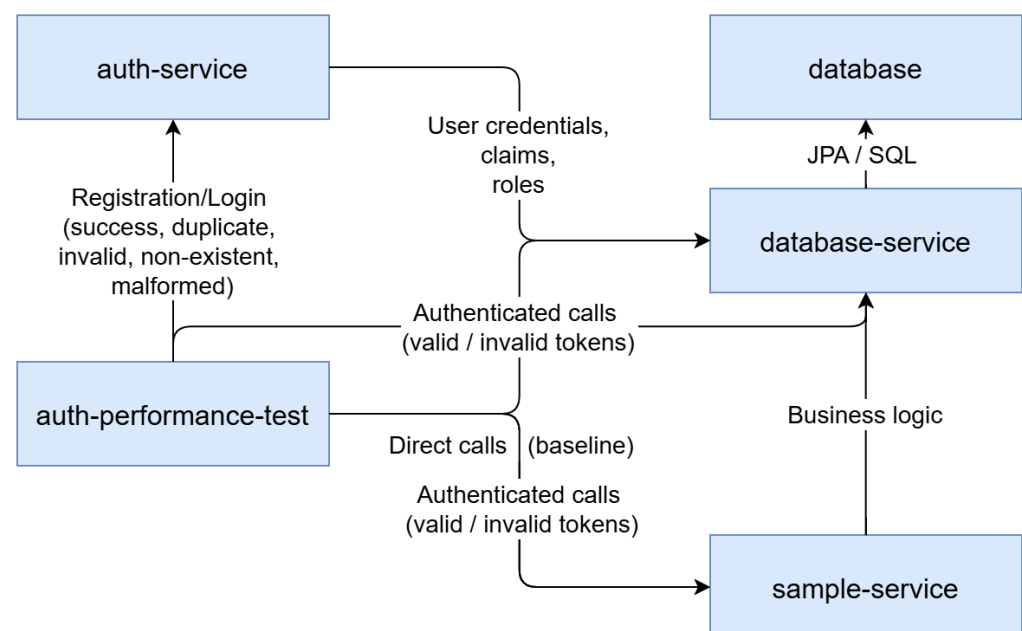


Figure 6. Test container flow.

The performance test container carries out a wide range of authentication and service interaction calls, including user registration and login scenarios, direct service requests, and authenticated requests with both valid and invalid tokens in order to measure system responsiveness and integration behavior.

When the auth-performance-test container is started, Docker ensures that all the dependent services are running. The test container then interacts with these services to simulate real-world scenarios and measure performance metrics. The results are saved to the results directory (Figure 7).

Two categories of performance test were conducted to evaluate the robustness and efficiency of the authentication and service components. The authentication performance tests comprised 1000 successful authentication requests, including 500 user registrations and 500 logins, and 1000 failed authentication attempts to simulate various failure scenarios. These tests also measured the performance of JWT token validation. Meanwhile, the Service Comparison Tests assessed the impact of authentication on service performance by executing 2000 unauthenticated service calls and 2000 authenticated calls using JWT tokens. Additionally, 500 requests with invalid tokens were tested to evaluate error handling and security. The results enabled a comparative analysis of response times and authentication overhead.

As Table 9 shows, analyzing authentication performance reveals notable differences between successful logins and failed authentication attempts. Successful logins have an average response time of 197 ms, which is slower than the 23–34 ms range for failed attempts. Failed authentications, whether due to invalid credentials, non-existent users, or malformed requests, are usually rejected early in the process, resulting in faster response times. These findings emphasize the efficiency of failure handling mechanisms and the additional costs associated with secure login operations.

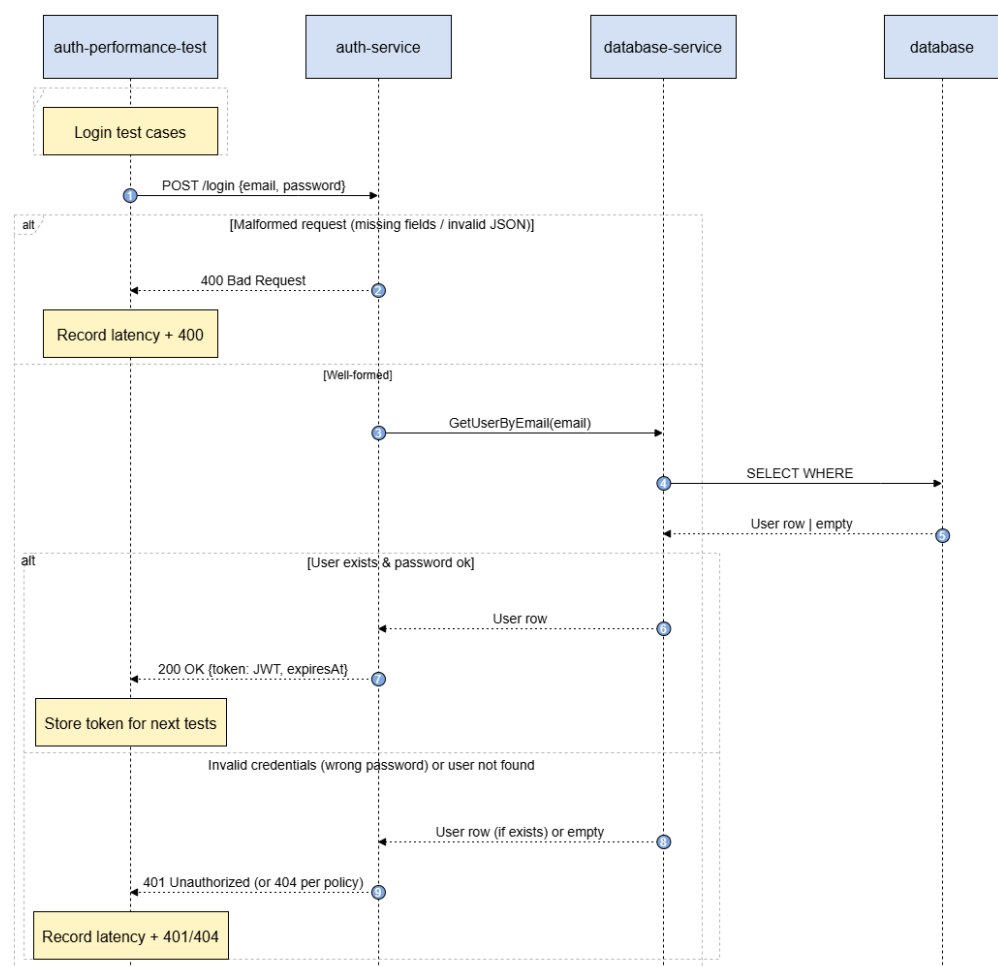


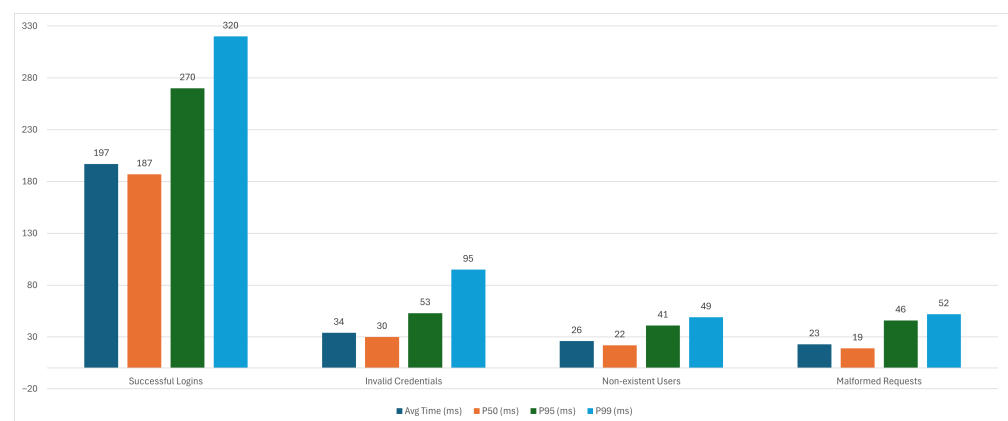
Figure 7. Instrumentation points in login flow.

Table 9. Comparison of response times for successful logins vs. failed authentication attempts.

Scenario	Count	Avg Time (ms)	P50 (ms)	P95 (ms)	P99 (ms)
Successful Logins	500	197	187	270	320
Invalid Credentials	400	34	30	53	95
Non-existent Users	300	26	22	41	49
Malformed Requests	200	23	19	46	52

The response time percentiles (P50, P95, and P99) offer useful information about the system’s performance in various authentication scenarios (see Figure 8). For successful logins, the P50 value of 187 ms indicates that half of the requests are completed within this time. Moreover, the P95 and P99 values (270 ms and 320 ms, respectively) reveal a noticeable tail in the latency distribution. This variability in processing times is likely due to backend operations such as credential verification and JWT token generation.

In contrast, failed authentication attempts (invalid credentials, non-existent users, and malformed requests) exhibit significantly lower and more consistent response times. For instance, invalid credentials have a P50 of 30 ms and a P99 of 95 ms, while non-existent users and malformed requests show even tighter distributions, with P99 values of 49 ms and 52 ms, respectively. These narrow ranges suggest that failures are quickly detected and handled without extensive processing, indicating efficient short-circuiting in the authentication logic.

**Figure 8.** Authentication response time comparison.

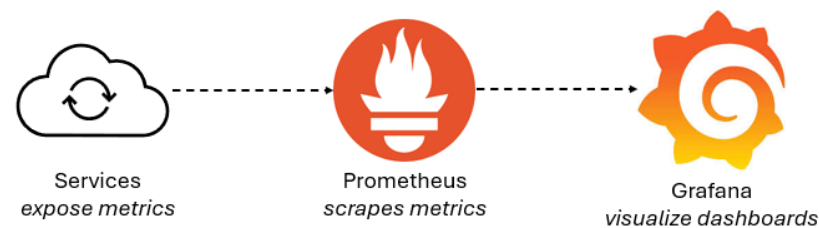
In summary, the percentile data highlights that the system is optimized for the rapid detection of failures, while successful authentications, albeit slower, remain within acceptable performance thresholds.

Table 10 shows that, for the *Sample Service*, for example, authenticated requests demonstrated better performance than direct calls, with an average response time of 23 ms compared to 32 ms. This improvement was consistent across percentiles, with P50, P95, and P99 values all being lower in the authenticated scenario. This suggests that authentication may enable optimizations such as caching or streamlined processing. In contrast, the *Database Service* showed consistent performance for both access types, with an average response time of 13 ms, and minimal variation in the P95 and P99 values. These results imply that the *Database Service* is highly optimized and unaffected by authentication overhead. However, further investigation is needed to establish why unauthenticated calls are slower for the *Sample Service*. This could be due to differences in request handling or resource access patterns.

Table 10. Performance metrics for direct vs. authenticated service calls.

Service Type	Service	Count	Avg Time	P50	P95	P99
Direct (No Auth)	sample	1000	32	25	75	168
	db	1000	13	12	21	52
Authenticated (JWT)	sample	1000	23	18	42	61
	db	1000	13	12	22	27

Furthermore, an empirical approach to enhancing performance evaluation is to use Prometheus, an open-source system that collects and stores time-series metrics from applications and infrastructure, to gather data from defined instrumentation points, and visualize these metrics through Grafana dashboards, which provide interactive and customizable views (Figure 9). This setup enables real-time monitoring of login performance and error rates.

**Figure 9.** Visualize metrics flow.

4. Conclusions

The interconnection of microservices, with the use of security controls to authenticate communicating entities (users or automated services) is necessary to protect data. Implementing authentication and authorization strategies, using API gateways, and adopting monitoring practices ensure that interactions between applications remain secure.

Using the case study and scientific literature, we analyzed how zero trust principles unify and enforce critical countermeasures against security flaws in microservices, particularly in the IAM subsystems of a microservice ecosystem. This ensures that all access, interactions, and data flows are continuously verified and operate with the least possible privilege, making the system resilient to breaches.

Nevertheless, for the proof-of-concept implementation on which the paper is based, the performance of the authentication process was evaluated using 500 successful logins and 1000 failed attempts. The latency percentiles (P50, P95, and P99) and average response times were reported. Successful logins averaged 197 ms (P95: 270 ms; P99: 320 ms), whereas failed attempts averaged 23–34 ms, demonstrating efficient error handling.

The emergence of quantum technologies and AI agents implies the need for continuous improvement of authentication methods and underlying encryption algorithms, and the creation of architectures that can prevent denial-of-service attacks in their early stages. Based on the concept of microservice-oriented architectures analyzed in the paper, issues that may affect the security of distributed systems are identified, and possible countermeasures are discussed.

Future research could examine how machine learning-based intrusion detection systems could be used to improve security in microservice architectures. Additionally, behavior analysis techniques could be investigated to identify and respond to unusual user or system activity. Another approach to proactive security management involves adaptive IAM policies that dynamically adjust access controls in response to detected anomalies.

Author Contributions: Conceptualization, C.L.A. and R.B.; methodology, C.L.A. and R.B.; software, C.L.A. and R.B.; validation, C.L.A.; formal analysis, R.B.; investigation, C.L.A.; resources, C.L.A. and R.B.; data curation, R.B.; writing—original draft preparation, C.L.A. and R.B.; writing—review and editing, R.B. and C.L.A.; visualization, C.L.A. and R.B.; supervision, R.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: Some English phrases corrected with <https://www.deepl.com/en/write> (accessed on 25 September 2025).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACM	Access Control Mechanism
CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
DAC	Discretionary Access Control
DoD	US Department of Defense
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
HSTS	HTTP Strict Transport Security
IAM	Identity and Access Management
IAMaaS	IAM as a Service
IdM	Identity management
IoT	Internet of Things
JAAS	Java Authentication and Authorization Service
JWT	JSON Web Tokens
LDAP	Lightweight Directory Access Protocol
MAC	Mandatory Access Control
MFA	Multi-Factor Authentication
mTLS	mutual Transport Layer Security
OAuth	Open Authorization
OIDC	OpenID Connect
OTC	One Time Code
OWASP	Open Worldwide Application Security Project
PAC	Privileged Access Management
PSA	Perimeter-based Security Architecture
RBAC	Role-Based Access Control
SIM	Subscriber Identity Module
SAML	Security Assertion Markup Language
SBOM	Software Bill of Materials
SECaaS	Security as a Service
SGX	Intel Software Guard Extensions
SMS	Short Message Service
SSO	Single Sign-On
ZT	Zero Trust
ZTA	Zero Trust Architecture

References

1. Nasab, A.R.; Shahin, M.; Raviz, S.A.H.; Liang, P.; Mashmool, A.; Lenarduzzi, V. An empirical study of security practices for microservices systems. *J. Syst. Softw.* **2023**, *198*, 111563. [\[CrossRef\]](#)
2. Sharma, D.H.; Dhote, C.; Potey, M.M. Identity and access management as security-as-a-service from clouds. *Procedia Comput. Sci.* **2016**, *79*, 170–174. [\[CrossRef\]](#)
3. Arif, T.; Jo, B.; Park, J.H. A Comprehensive Survey of Privacy-Enhancing and Trust-Centric Cloud-Native Security Techniques Against Cyber Threats. *Sensors* **2025**, *25*, 2350. [\[CrossRef\]](#) [\[PubMed\]](#)
4. Ometov, A.; Petrov, V.; Bezzateev, S.; Andreev, S.; Koucheryavy, Y.; Gerla, M. Challenges of multi-factor authentication for securing advanced IoT applications. *IEEE Netw.* **2019**, *33*, 82–88. [\[CrossRef\]](#)
5. Mateus-Coelho, N.; Cruz-Cunha, M.; Ferreira, L.G. Security in microservices architectures. *Procedia Comput. Sci.* **2021**, *181*, 1225–1236. [\[CrossRef\]](#)
6. Xu, S.; Yin, M.; Hu, T.; Li, J.; Li, H.; Deng, K. Traffic Verification Based on API Function Control in IIoT Service Cluster. In Proceedings of the 2023 IEEE 11th International Conference on Information, Communication and Networks (ICICN), Xi'an, China, 10–13 August 2023; IEEE: New York, NY, USA, 2023; pp. 347–352.
7. Chatterjee, A.; Prinz, A. Applying spring security framework with KeyCloak-based OAuth2 to protect microservice architecture APIs: A case study. *Sensors* **2022**, *22*, 1703. [\[CrossRef\]](#) [\[PubMed\]](#)
8. RFC 7519; JSON Web Token (JWT). Available online: <https://datatracker.ietf.org/doc/html/rfc7519> (accessed on 10 November 2025).
9. Darmawan, I.; Karim, A.P.A.; Rahmatulloh, A.; Gunawan, R.; Pramesti, D. Json web token penetration testing on cookie storage with csrf techniques. In Proceedings of the 2021 International Conference Advancement in Data Science, E-Learning and Information Systems (ICADEIS), Nusa Dua, Indonesia, 13–14 October 2021; IEEE: New York, NY, USA, 2021; pp. 1–5.
10. Chulhyung, L.; Fiona Kim, H.; Lee Bong, G. A Systematic Literature Review on the Strategic Shift to Cloud ERP: Leveraging Microservice Architecture and MSPs for Resilience and Agility. *Electronics* **2024**, *13*, 2885. [\[CrossRef\]](#)
11. Pu, Z. Smart Campus Business Flow Reconstruction Algorithm Based on Unified Identity Authentication and Microservices. In Proceedings of the ICSETPSD 2023: The First International Conference on Science, Engineering and Technology Practices for Sustainable Development, ICSETPSD 2023, Coimbatore, India, 17–18 November 2023.
12. Bilal, O.; Raza, A.; Rehman Khan, S.U.; Ali, G. A Contemporary Secure Microservices Discovery Architecture with Service Tags for Smart City Infrastructures. *VFAST Trans. Softw. Eng.* **2024**, *12*, 79–92. [\[CrossRef\]](#)
13. El Akhdar, A.; Baidada, C.; Kartit, A.; Hanine, M.; Osorio Garcia, C.; Garcia Lara, R.; Ashraf, I. Exploring the potential of microservices in internet of things: A systematic review of security and prospects. *Sensors* **2024**, *24*, 6771. [\[CrossRef\]](#)
14. Atalay, T.; Maitra, S.; Stojadinovic, D.; Stavrou, A.; Wang, H. An OpenRAN Security Framework for Scalable Authentication, Authorization, and Discovery of xApps With Isolated Critical Services. *IEEE Trans. Dependable Secur. Comput.* **2025**, *22*, 2873–2890. [\[CrossRef\]](#)
15. Palavali, D.R. Agentic AI for Self-Sovereign Identity: A Decentralized Zero Trust Framework for Autonomous Microservices. *Int. J. Comput. Math. Ideas IJCM* **2025**, *17*, 17347–17360.
16. Nahar, K.; Gill, A.Q. Integrated identity and access management metamodel and pattern system for secure enterprise architecture. *Data Knowl. Eng.* **2022**, *140*, 102038. [\[CrossRef\]](#)
17. Stafford, V. *Zero Trust Architecture*; NIST Special Publication 800-207; NIST: Gaithersburg, MD, USA, 2020.
18. He, Y.; Huang, D.; Chen, L.; Ni, Y.; Ma, X. A survey on zero trust architecture: Challenges and future trends. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, 6476274. [\[CrossRef\]](#)
19. Lopez, J.; Oppliger, R.; Pernul, G. Authentication and authorization infrastructures (AAIs): A comparative survey. *Comput. Secur.* **2004**, *23*, 578–590. [\[CrossRef\]](#)
20. Shah, S.W.; Kanhere, S.S. Recent trends in user authentication—A survey. *IEEE Access* **2019**, *7*, 112505–112519. [\[CrossRef\]](#)
21. Eckert, C. *IT-Sicherheit 7. A. Konzepte–Verfahren–Protokolle*; Oldenbourg Wissenschaftsverlag: Munchen, Germany, 2012.
22. Gunson, N.; Marshall, D.; Morton, H.; Jack, M. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *Comput. Secur.* **2011**, *30*, 208–220. [\[CrossRef\]](#)
23. Syed, N.F.; Shah, S.W.; Shaghaghi, A.; Anwar, A.; Baig, Z.; Doss, R. Zero trust architecture (zta): A comprehensive survey. *IEEE Access* **2022**, *10*, 57143–57179. [\[CrossRef\]](#)
24. Aldea, C.L. *Data Security Elements in Computer Networks*; Transilvania University of Brasov Publishing House: Brasov, Romania, 2010.
25. Indu, I.; Anand, P.R.; Bhaskar, V. Identity and access management in cloud environment: Mechanisms and challenges. *Eng. Sci. Technol. Int. J.* **2018**, *21*, 574–588. [\[CrossRef\]](#)
26. Calhoun, Z.; Maribojoc, P.; Selzer, N.; Procopi, L.; Bezzo, N.; Fleming, C. Analysis of Identity and Access Management alternatives for a multinational information-sharing environment. In Proceedings of the 2017 Systems and Information Engineering Design Symposium (SIEDS), Charlottesville, VA, USA, 28 April 2017. [\[CrossRef\]](#)

27. Ibrokhimov, S.; Hui, K.L.; Al-Absi, A.A.; Lee, H.J.; Sain, M. Multi-factor authentication in cyber physical system: A state of art survey. In Proceedings of the 2019 21st International Conference on Advanced Communication Technology (ICACT), Pyeongchang, Republic of Korea, 17–20 February 2019; IEEE: New York, NY, USA, 2019; pp. 279–284.
28. Spring Boot Building an Application with Spring Boot. 2025. Available online: <https://spring.io/guides/gs/spring-boot> (accessed on 12 October 2024).
29. Ponce, F.; Soldani, J.; Astudillo, H.; Brogi, A. Smells and refactorings for microservices security: A multivocal literature review. *J. Syst. Softw.* **2022**, *192*, 111393. [[CrossRef](#)]
30. Tourchi Moghaddam, M.; Edal Pedersen, A.; Lillebroe Bolding, W.W.; Worm, T. A Performant and Secure Single Sign-On System Using Microservices. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, Tallinn, Estonia, 27–31 March 2023; pp. 1516–1519.
31. Woschek, M. *Owasp Cheat Sheets*; OWASP Foundation: Wilmington, DE, USA, 2015; pp. 1–315.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.