International Journal of Leading Research Publication (IJLRP)
E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Blockchain-Enabled Multi-Cloud Framework for Gemstone Certification and Traceability

Sashi Kiran Vuppala

McKinney, Texas, USA
sashivuppala93@gmail.com

Abstract

In the era of cloud-native applications, microservices have emerged as a dominant architectural paradigm for building scalable and modular software systems. However, with this flexibility comes significant challenges in maintaining robust security and comprehensive observability. This paper presents a unified, microservices-based framework for securing and monitoring gemstone certification workflows by integrating blockchain technology, AI-based analytics, and a multi-cloud orchestration model. The proposed architecture enforces strict access controls using JWT to authenticate certification requests and validate traceability records. To further tighten security, Zero Trust is implemented to eliminate implicit trust in network boundaries, requiring continuous validation of every request regardless of source. On the observability front, logs, metrics, and traces are collected and centralized using Splunk, providing developers and administrators with actionable insights into system performance and anomalies. Performance metrics such as authentication success rate, response time, error rate, and system health score are evaluated before and after integration. The results show a marked improvement in security enforcement and operational transparency. JWT authentication success rates increased, response times decreased, and system health scores improved, demonstrating the positive impact of the integrated framework. This research contributes a practical methodology for organizations aiming to build secure, observable microservices systems. By addressing both security and traceability in a unified flow, the framework supports early detection of anomalies and operational inefficiencies, allowing proactive system management. The study also outlines a scalable path forward by suggesting future integration with AI-based quality assessment, mobile verification tools, and blockchain scalability enhancements. Overall, the paper offers a robust model suitable for modern distributed application environments.

Keywords: Microservices, Blockchain, Multi-Cloud Architecture, Gemstone Certification, Traceability, Hyperledger Fabric, Smart Contracts, AI-based Authentication, JWT Authentication, Zero Trust Security, Splunk

I. INTRODUCTION

Microservices have become typical today in new software development as an aspect of the basic architectural design for applications to be scalable, modular, and agile[1]. This is one big step away from traditional monolithic systems that bundled functionalities in a single codebase. Decomposing an application into loosely coupled units that can be independently deployed and serviced- define

microservices architecture. These services are specific to business needs and communicate through simple, lightweight protocols like HTTP/REST or gRPC. Architectural change drives optimum Continuous delivery, quicker iteration, and horizontal scaling which are today very critical for cloud-native and DevOps features. However, the scale of complexity increases manifold due to interservice communication and runtime observability. Java has been the go-to language for enterprise microservices due to its mature ecosystem, the presence of substantial frameworks such as Spring Boot and Micronaut, and widespread support for containerization and orchestration platforms such as Docker and Kubernetes[2]. But still, the developers are facing yet another increased challenge in securing microservices from internal and external threats, providing full visibility among service boundaries. These demands have stimulated the necessity of integration of smart observability tools and advanced security models specifically designed for distributed environments.

Security in microservices goes way beyond traditional firewall and network perimeter protections; it requires us to rethink security as a dynamic, embedded concern at the service level[3]. In contrast to monolithic systems, which tend to rely on trust between internal components by default, microservices could be deployed in untrusted environments, especially within hybrid or multi-cloud infrastructures. Therefore, each microservice has its attack surface and may be independently vulnerable, including being attacked, unless subject to uniform service-oriented security policies. To this end, JSON Web Tokens (JWTs) are crucial for the security of microservices because they provide both stateless authentication and fine-grained authorization. In this manner, JWTs encapsulate the user identity, roles, and claims within a cryptographically signed token that can be verified by any service without keeping the session state. That's how security is maintained as a key area of focus for horizontally scalable Java-based microservices, where each instance can independently validate JWTs. Used with appropriate expiration, signature verification, and role-based access control (RBAC), the JWTs prevent unauthorized access, reduce the risk of session hijacking, and allow the working of zero trust principles[4]. However, libraries in Java such as jjwt, Nimbus JOSE + JWT, and support in frameworks like Spring Security, make it easy to put JWTs in the authentication flow. As APIs keep growing and microservices get ever more complicated concerning service meshes, JWT-based security will ensure.

In an environment distributed through microservices, observability remains the necessary base for operational stability, quick incident response, and for the sake of improvement[5]. Unlike monolithic applications, observability becomes a key requirement in microservices because the application straddles across containers, nodes, and sometimes even cloud providers. It thus becomes extremely difficult to keep track of where failures originate, how performance is monitored, or even how threats are detected; much, therefore, depends on varied observability strategies[6]. For example, Splunk presents an enterprise solution for the ingestion of logs, metrics, and traces of a variety of microservices into one center point for clear visibility into the very state of the system for administrators, SREs, and developers. For Java applications, Splunk's support of structured logs and its log parsers allow the easy ingestion, rich indexing, and real-time analysis of logs generated by frameworks like Spring Boot, Logback, and Log4j. With dashboards, alerts, and insights into the historical data, teams can quickly detect anomalies, trace request failures, measure service latency, and correlate disparate events in the system that otherwise would have been impossible to understand in isolation. This leads to not only reducing the mean time to resolution (MTTR) during incidents but also proactively tuning of the services based on trends and usage.

Splunk's technical integration into Java-based microservices involves, quite honestly, more than just log collection builds an observability fabric that allows traceability, auditability, and operational security[7]. Developers can configure HTTP Event Collectors (HECs) to stream logs from services directly to Splunk in near real-time while also complying with data retention and governance policies. OpenTelemetry can be utilized for distributed tracing and span information to provide a richer context for developers to identify and replicate the entire request journey across services. Along with correlation IDs injected in logs and headers, this makes performance bottlenecks and cascading failures quite easier to trace. Splunk dashboards could also be customized to surface critical security telemetry, including JWT token validation failure, unusual authentication attempts, and access control violations so that security and observability could sit under one roof[8]. Especially when matched with Zero Trust principles, while Scooping up anything remotely related to these in security and observability is bearing lots more teeth unto Splunk. For, each access request and service communication can thus be scrutinized, verified, and recorded to become an immutable audit trail for regulatory compliance and threat hunting. Security policies, once integrated with a well-implemented observability system within the microservices layer, will help the organization build resilient, transparent, and highly secure Java cloud-native applications ready to run in hybrid environments.

Zero Trust is certainly not a security ducttape clause but a statement of a revolution, redefining how modern-day distributed systems will establish and maintain trust. Security models relying on the invalid assumption that entities benignly residing within the network perimeter deserve that trust have probable cause to revise those assumptions. In microservices-based architectures, which may span several networks, data centers, and cloud environments, this assumption is indeed dreadfully dated[9]. To ensure that entities and interactions within the system are continuously authenticated, authorized, and validated, Zero Trust adopts the attitude of "never trust, always verify." There are several layers in the application of Zero Trust in Java-based microservices. First, mutual TLS (mTLS) can be enforced between services to establish a cryptographically secure communication channel such that only verified microservices are allowed to exchange data. Second, the use of JWTs with fine-grained scopes and roles allows for stateless, identity-aware access control, enabling services to validate the authenticity and privileges of each request without the need to query a central authority. Third, each incoming internal request will undergo integrity checks, token validation, and policy enforcement before the execution of any sensitive operation[10]. The mechanisms complement one another to interfere with lateral movement across the network while containing the breach, further facilitating real-time access governance in cloud-native environments, extremely dynamic.

With the introduction of Zero Trust combined with JWT for authentication and Splunk for observability, organizations can achieve an excellent trifecta of protection, visibility, and control within their entire microservices ecosystem. Each module strengthens the others; JWT is the mechanism for secure state-less identity assertion, while Splunk gives teams deep operational insight into authentication patterns, interactions between services, as well as anomalies; and finally, Zero Trust dictates the major domain of strict, continuous validation. Together, they create a proactive rather than reactive security architecture that can in real-time identify unauthorized access attempts, enforce least privilege principles network-wide, and then create complete audit trails for compliance and forensics. In Java applications, such integration can be done with Spring Security for JWT validation, Envoy or Istio for mTLS and policy enforcement, and Splunk using Open Telemetry for placing all observability in a single place.

Such security by design allows developers and DevOps teams to build scalable yet secure, agile yet governed systems. Integrated models into microservices architectures are very robust concerning the future and against an ever-evolving threat landscape, in the age of ever-growing sophisticated cyber threats and regulations on what data can be accessed.

The Key contributions of the article are given below,

- Proposed an integrated architecture that combined JWT authentication, Zero Trust security principles, and Splunk-based observability to enhance both security and monitoring in Java microservices.
- Implemented JWT to ensure secure, stateless authentication across distributed services, effectively reducing unauthorized access and token misuse.
- Applied Zero Trust principles to enforce strict identity verification for every request, regardless of its origin, thereby eliminating implicit trust within the network.
- Integrated Splunk for centralized logging and observability, enabling real-time performance tracking, anomaly detection, and system diagnostics.
- Validated the effectiveness of the approach through empirical evaluation, showing measurable improvements in authentication success rate, error reduction, system health,and response times.

This document is organized as follows for the remaining portion: Section II discusses the related work. The recommended method is described in Part III. In Section IV, the experiment's results are presented and contrasted. Section V discusses the paper's conclusion and suggestions for more study.

II. Related Works

A. Role of Microservice

In this study by Monteiro et al.[11], an adaptation for an observability technique based on the game theory is proposed to be able to prepare digital forensics with the microservice-based applications for which ephemeral containerized infrastructures pose great difficulties when it comes to preserving digital evidence during security incidents. Traditional observability techniques do not go well with forensic investigations—particularly concerning challenges that are either unforeseen or unpredictable. The proposed approach constantly adapts to uncertainties in its simulations of user-microservice interactions to figure out optimal monitoring and evidence-gathering strategies before catastrophic events take place. The evaluations indicate that the proposed adaptive technique can achieve forensic readiness improvements between 3.1% and 42.5%, which is largely ahead of the contemporary observability approaches.

Schneider et al.[12] provide explanations that are transparent and traceable, thus linking verdicts to actual source code artifacts by expressing security rules and evaluating them as model queries using a custom-defined rule specification language. The approach supports both user-defined rules and standard best-practice rules. High accuracy (precision=0.98; recall=1) was obtained in the experimental quantitative test, which was conducted on 16 data flow diagrams. The utility, usability, and integration potential were corroborated via qualitative feedback from six industry experts commenting on the workflow for routine usage of this tool in security assessment processes.

B. Microservice Architecture

Interoperability in IoT involves the seamless communication of these various devices, embedded with different data formats, interworking in different ecosystems is studied by (Vila Gómez, Sancho Samsó, and Teniente López[13]. Therefore, the authors have proposed an ontology for sensor, actuator, and real-world data that will standardize and contextualize information necessary for event-driven rules automation for smart, automated, and intelligent monitoring. This ontology can be a competing framework for use in heterogeneous systems worldwide to interpret and exchange information. It is a cloud solution provided as a microservice architecture for the collection and monitoring of real-time data, with automatic responses being part of the overall solution aimed at fostering interoperability, adaptability, and real deployments across different sectors.

Poghosyan et al.[14] studied that DL algorithms are inherently good at detecting fraud or assessing risks in real-time and thus can be brought into play for the larger SaaS domain of Cloud Payment Infrastructure. One has to predict with a technology that can scale and maintain low latencies and enormous throughputs while riding the waves of rapidly changing transactional patterns. CI/CD pipelines have to be in place to ensure that models are capable of changing regularly because the latest architecture of transaction-to-fraud requires it due to globally distributed complexity, changing fraud strategies, and disparate compliance requirements. This research focused on scalable deployment avenues that ensure performance and regulatory compliance, studied deep learning techniques for anomaly detection, and outlined some key architectural principles for SaaS platform(s). It also stresses the importance of the system security and the interpretability of the model, while discussing the feasibility and long-term implications of deep learning for dynamic financial risk management.

C. Advantages of Microservices

Ding et al.[15]present an end-to-end Root Cause Analysis (RCA) framework aimed at improving dependability in large-scale microservices systems with a functional view toward potentially addressing complexities and human work often involved. Through reinforcement training, TraceDiag creates a pruning policy to intelligently prune the service dependency graph to get rid of unnecessary elements that clod the effectiveness and interpretability of RCA. Causal analysis can then take place faster and with more accuracy. TraceDiag has been integrated with Microsoft M365 Exchange, where it was evaluated with actual traces processed in Microsoft Exchange. It outperforms existing RCA methods to enhance system dependability and reduce the manual effort for RCA.

Niknejad et al. [15]deal with the design and implementation of an actual software system that combines a Machine Learning-As-A-Service for upgrading a remote surveillance framework. The redesign was done using microservices architecture to allow for easy scalability and gradual uptake. Machine learning thus could lend itself to event management through backend data-stream processing. A case study illustrating the successful integration of the surveillance systems highlighted a reduction in information overload for human operators, thereby enhancing their capability to handle and respond to incidents in surveillance operations.

D. Microservices for Cloud& Blockchain Integration

Berardi[16] studied that from massive data centers to mission-critical infrastructures, in this dissertation, we are going to see network technology changes, especially the security implications of

advances like SDN and the Industry 4.0(I4.0) movement. With inadequate security, a traditionally isolated industrial network becomes increasingly vulnerable to hackers as it connects to the internet for data-driven capabilities like predictive maintenance. This thesis presents an exploration of the offensive and the defensive, showcasing real-world attack scenarios on vulnerable new infrastructures and providing recommendations for defense, monitoring, and detection exploits using these next-generation network technologies. Three years of studies are summarized, aiming at uncovering security flaws and providing viable solutions drawn from solid analyses, with a focus on several actual case studies.

XSS and SQL injection attacks are well-known as troublesome security issues, and they become even more complicated in a cloud-based microservices setting than in the highly simplified regular monolithic application systems and it is studied by Joshi [17]. Due to the unavailability of labeled intrusion datasets, the paper applies unsupervised machine-learning-based intrusion detection on a microservices platform organized around Kubernetes, with more than 40 services in place. The system characterized baselines derived from its performance metrics and service logs for normal application behavior and then searched for anomalies supporting indicators of possible attacks. The key contributions are the use of unsupervised machine-learning algorithms for intrusion detection in microservice architectures and the fusion of profiling application performance data alongside log analysis.

III. Research Methodology

A. Research Gap

Despite increasing concerns over microservices-based applications in the cloud, the existing intrusion detection methods suffer from several major limitations which create a clear research gap. The traditional approaches are primarily monolithic by design, based on signature or some supervised learning paradigm that requires a very large labeled dataset-never available in a realistic microservices situation[18]. These techniques do not offer the scalability or adaptability to the decentralized, dynamic, and ephemeral properties of microservices where services are continuously being enhanced and update-deployed independently-generate massive volumes of heterogeneous logs and performance data. Most current IDSs further fail to integrate multiple data sources, like performance metrics, and logs, which would enhance detection capabilities for services under complex, low-and-slow attacks[19]. Tools that do consider metrics often ignore the contextual and semantic richness of the logs themselves, leading to an increase in false positives or, worse, missed detections. Also, they lack adaptability and respond in realtime, which is required in microservice ecosystems orchestrated by a platform like Kubernetes, where services can quickly scale up or down. Further, in many studies detection has been focused solely onnetwork and not on application-level anomalies that are key in detecting sophisticated web-based attacks like XSS or SQL Injection. This leaves an enormous unsupervised research gap.

B. Proposed Framework

Within the architecture proposed by unified secure monitoring of Java microservices, the core components of JWT Authentication, Zero Trust Principles, Splunk Integration, Observability Tools, Monitoring & Alerting come togetheras shown in Fig 1. This highlights the integration through which Splunk is made central to the architecture, emphasizing how it takes responsibility for all logs collected from various services to provide insight. JWT Authentication secures access by issuing and verifying tokens, while Zero Trust Principles further demand that every request be explicitly verified. The

observability tools work together with Monitoring & Alerting to form a real-time performance insight and anomaly detection within a bigger picture for proactive system health management as a whole. This whole figure presents a layered model of security and insight-driven governance for resource-efficient management in a distributed Java microservices environment.

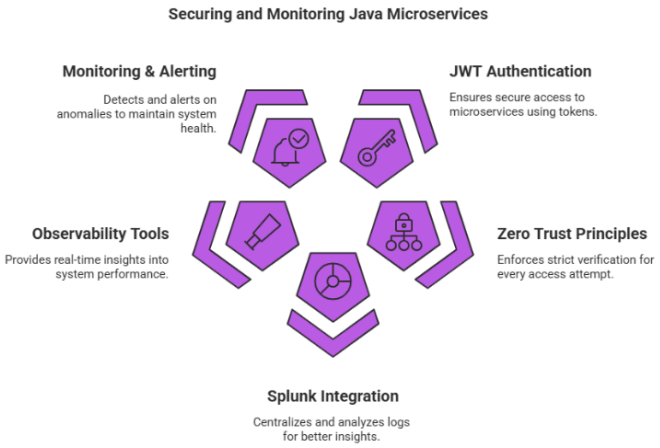

Fig. 1. Proposed Framework

c. Set Up Java Microservices

Moving to Java microservices would mean strategically breaking down a monolithic application into different modular and independently deployable services according to distinct business capabilities. Such a shift in architectural design allows for the scalability, fault isolation, and empowerment of teams. Each microservice works by itself, loosely coupling, and they communicate with each other over lightweight protocols like RESTful APIs over HTTP or through asynchronous messaging technology such as Apache Kafka or RabbitMQ. The first step of microservices setting is the framework selection among Spring Boot, Micronaut, or Quarks typically those frameworks that exercise proven productivity increase by giving you an out-of-the-box experience on an embedded server, key injection of an application's components and secure REST endpoints access. After the business definition is encapsulated for each microservice it has been docker and orchestrated by Kubernetes or Docker Swarm to achieve elastic scalability, high availability, and resource-efficient use across distributed environments.

One of the most important architectural designs in microservices is a clear definition of service boundaries and articulation of APIs, which define how services interact with each other. Each microservice should encapsulate only one domain responsibility so that it can separate the concerns and reduce interdependencies. RESTful APIs are usually well-defined and documented with tools like OpenAPI (which used to be known as Swagger), thereby ensuring consistent communication, driving development based on contracts, and facilitating clients' integration. Furthermore, such mechanisms as Netflix Eureka, Consul, or native Kubernetes DNS-based discovery would be used to dynamically and robustly communicate in a distributed system. This would allow services that would have been hard-coded not to be able to find each other another, which would be essential in the case of cloud-native deployments, where services tend to scale up or down or restart for some reason. Additional

microservices call for externalized configuration, centralized logging, and standard error handling to guarantee maintainability and observability from the get-go; this lays the foundation for further secure and reliable operations in the system.

All of the services in a microservices setup must, by design, incorporate fault tolerance and resiliency as important aspects. Such features would include structured exception handling, timeout settings that can be configured, and smart retry techniques that try to handle nonpersistent failures gracefully. Circuit-breaker integration—using libraries like Resilience4j or the now-defunct Hystrix—takes this further towards a functional resilience scenario by preventing the rest of the system from breaking down like a house of cards if one of its dependent services fails to respond. Such patterns serve to isolate the unresponsive components and allow the remaining working components of the system to keep working while they implement failover strategies. Some cross-cutting concerns are configuration management, rate limiting, load balancing, and authentication: with tools like Spring Cloud Config for distribution configurationor API gateways such as Zuul, Kong, or API Gateway in cloud-aware platforms-it is abstracted into a unified center. More importantly, it centralizes these concerns for consistency, security, and maintainability across services while avoiding redundant code and manual synchronization.

Another core principle of Java microservices setup is maintaining data isolation via the database-per-service paradigm, which requires each microservice to have an independent data source. This encourages loose coupling with the underlying data layer, allowing teams to evolve, tune, or scale services independently without inflicting cross-service database conflicts. Microservices have been packaged inside containers using Docker, which provides isolated and reproducible runtime environments. The containers go into integration into CI/CD pipelines, managed by tools like Jenkins, GitHub Actions, or GitLab CI. The pipelines allow for the automation of code compilation, testing, security checks, and deployment, which helps release faster and with more confidence. Kubernetes manages the deployment, scaling, and operations of these containers, providing features for self-healing, rolling updates, and horizontal scaling. This modular container-native approach not only speeds development and deployment but also guarantees an immediate path to production and high availability, with secure monitoring and observability integrations.

D. Implement JWT-Based Authentication

Indeed, there is a composition that implements JWT authentication of microservices in Java. It should secure stateless authentication methods of user identity verification and access control among services deployed independently. Initially, the user needs to send a request with credentials to the authentication service, which usually checks them against a user database. Once proceeding through and validating the credentials, the authentication server creates and issues a JSON Web Token (JWT) to the end-user. A JWT consists of a header, containing information about how it has been signed; a payload, which has crucial claims that should be included in the token, like user ID, role, expiration of the token, and scope of access; and finally, an additional signature, using either a very secretive key or else public-private key pair (in case of asymmetric signing). The token is given to the client now, which needs to be present in the Authorization header of each further request to any microservice. Each service, thus, has middleware or filters usually implemented using Spring Security in Java, which allows intercepting incoming requests, extracting JWTs, validating their signatures, checking the claims (like expiration and audience), and deciding whether the request should be granted or blocked according to the permissions

embedded further therein. This way, servers do not maintain session information, making their performance better regarding high scalability in this model, particularly for disparate systems that are deployed in the cloud or using containerization within an organization. Moreover, it is used in fine-grained access control in which roles and permissions within the encoded JWT define much of the resource for all.

Stateless Authentication Mechanism

Stateless authentication is an important security concept in microservices architecture; it enables each service to independently authenticate and authorize requests without any tracking of session information. The main actor of this stateless model is JWT, whereby, upon successful login, the server generates a digitally signed token that encodes user claims such as user ID, roles, and expiration timestamp. This token is handed back to the client and added to the Authorization header of each subsequent HTTP request. In this scenario, the microservices that receive these requests extract and verify the token without the need for making calls to any central session store or performing repetitive database look-up calls for authentication. This saves on memory on the server side and allows more scalability in the distributed systems environment, especially when services are deployed in instances that are auto-scaled in containerized environments, such as Kubernetes.

Stateless authentication through using JWTs is another significant advantage concerning goals in horizontal scaling as well as high availability. Because user session information is embedded within a token, which is cryptographically signed by the server (usually through HMAC SHA-256 or RSA algorithms), any one instance of a microservice can now validate the authenticity of that token without having to look up any of that information on a memory-based or database session. Thus, any load-balanced instance can process a request without worrying about where the user's associated data might be hosted, ultimately maximizing computer resources and eliminating bottlenecks associated with session replication. In addition, the expiration time included in the JWT ensures automatic session timeouts, thus limiting the chance for random unauthorized long stays. In very secure environments, short-lived tokens may be supplemented with a refresh token and possibly strategies for revoking, so that they cannot be reused or abused.

On the operational side, stateless JWT authentication provides good observability and compliance. Therefore, token usage, validation results, and access decisions are logged by each service, which uses central logging platforms like Splunk. Thus, one can analyze access patterns in real-time, spot anomalies (e.g., token reuse or high validation failure rates), and provide audit trails for security compliance. Also, JWTs align with Zero Trust principles—which dictate that no user or device is trusted by default, not even one located within the network perimeter. By mandating that tokens be verified with every request, the architecture ensures continuous authentication and verification, thereby strengthening the overall security posture of the system. Visibility into, decoding of, and analysis of JWTs allow developers and security teams to gain additional insights into the authentication logic and user roles encoded within the tokens.

Token Validation in Microservices

Token validation for microservices is an essential component in ensuring that only authenticated and authorized users access the protected resources. Each microservice must validate the incoming token when an incoming request is received with JWT in the Authorization header before performing the requested operation. The overall validation encompasses multiple stages i.e. includes validity of cryptographic signature to claim that the token is coming from a trusted authority, expiration of the token to avoid stale credentials, and validity of claims in the token which are required for role-based control access checks. In such cases, within a Java microservices environment, especially with Spring Boot and Spring Security-based applications, the token filters and interceptors are used to parse and validate the JWT before it hits any service logic. Thus, each service makes real-time access-control decisions without centralization within an authentication server or session state for increased resilience and scalability to architecture.

Public key cryptography has been applied for token validation in this implementation, particularly for systems where one entity issues the token through a centralized identity provider and then multiple distributed services perform token validation. When asymmetric key pairs, such as RSA, are in use, the private key of the identity provider is used to sign the token, while the public key is used for validation by the microservices. Thus, there is no need for sharing any secret between the services, significantly reducing the risks of secret leakage. Furthermore, validation based on claims was enforced to verify required scopes, permissions, or roles, thus allowing for fine-grained authorization across microservices. The microservices will reject requests with an immediate response of 401 Unauthorized or 403 Forbidden if the token is not granted with certain claims or invalid data. The fine time control allows developers to specify access policies on a per-service endpoint level and prevents misuse by unauthorized personnel, even where a valid token is presented.

From security and observability perspectives, every token validation would be logged significantly to provide a complete owner audit trail into system usage. Each service will write a log regarding every attempt to validate the token, either as a success or failure. Other aspects such as expired or malformed tokens rejection and attempts to access unauthorized resources are also captured in the audit log. The centralization where logs are forwarded is at a centralized platform like Splunk aggregation for display and analysis towards insight. Security analysts can thus infer insights on these points to identify common spots where failures occur, determine brute-force attacks, and monitor trends in authentication over time. Not only that, but such a consistent token validation across microservices would also enforce the Zero Trust policy that every access request must go through the validation process before it can be awarded approval regardless of its source. Not only is it hardening the security perimeter but also visibility and traceability against any activity carried out in the layer of authentication, the importance being heightened in big Java applications deployed in a cloud-native environment.

Alignment with Zero Trust and Observability

JWT-based authentication is in line with Zero Trust principles, where the maxim runs: "Never trust, always verify." Users and devices within the perimeter are trusted under conventional security models. When this perimeter is breached, security cracks occur. Every requestthat comes from anywhereis treated as not trusted in ZTA, and it requires a strict validation process before access is granted. That is

something JWT would support because it would allow stateless, token-based verification at every microservice end-point. Microservices interact with each request, doing independent token validation rather than assuming the user's legitimacy based on prior authentication and enforcing constant authentication while minimizing lateral movement risks. Crucially, this takes the evolution from perimeter-based security into an identity-centric access-control model: the dynamic, real-time, and cryptographic integrity of retrospectively authenticated user claims and roles.

JWTs in this framework act not merely as access tokens but also serve as more than fine-grained authentication mechanisms encapsulating user metadata, roles, privileges, and expiration information into a secure self-contained structure. Microservices that work independently check these claims embedded in the tokens for access-level authorizations at the granularity of individual API routes. For instance, a certain service could allow specific operations such as deleting or modifying data restricted to tokens with admin roles, and the same service could allow all other tokens read-only access. This kind of dynamic and decentralized access enforcement strengthens the Zero Trust philosophy of least privilege. Another advantage of this is the possibility of denying tokens that come from untrustworthy sources or those that are signed with tampered signatures, thus reducing the possibility of forged or replay tokens. This kind of strategy makes the security paradigm more robust in that all decisions are based upon validated identity assertions rather than on static access control lists or assumptions based on network locations.

This integration between JWTs and centralized logging platforms like Splunk gives the organization a vantage point for authentication behavior monitoring, audit, and analysis at scale. Every token interaction generates telemetry: issuance, validation, expiration, or rejection. This telemetry is logged in a structured form that can later be exported, aggregated, and analyzed.This observability layer will allow security and operations teams to visualize patterns in authentication, identify anomalies (e.g., repeated invalid token attempts or unusual access locations), and enforce alerting policies for real-time incident response. The metadata inside JWTs—such as user ID, IP address, scopes, expiration, etc.—greatly richens this traceability context across microservice transactions for faster debugging, forensic investigations, and compliance reporting. This is how tying JWT usage to observability pipelines goes beyond mere access control to become a proactive, insight-driven security framework—providing the fullest demonstration of the synergy between Zero Trust enforcement with operational transparency.

The authentication process using JWT is illustrated in Fig 2, where an important note is made about security between client-server-database in a microservices architecture. It mainly starts when a client gives user credentials to the server to prove himself. After inferring whether the credentials are valid through verifying with the database, the server generates a JWT. This token is generated with some identity and access claims associated with a user. The generated token is returned to the client, which will append it to every request for the server to validate and authorize the client's access to the protected resources. The server accepts the JWT with each request and does not check credentials, supporting a stateless and hence scalable method for its authentication. The diagram here strengthens the security model adopted in this study as it brings focus on the efficiency and security of access control that gives life to token traceability under observability and Zero Trust principles.

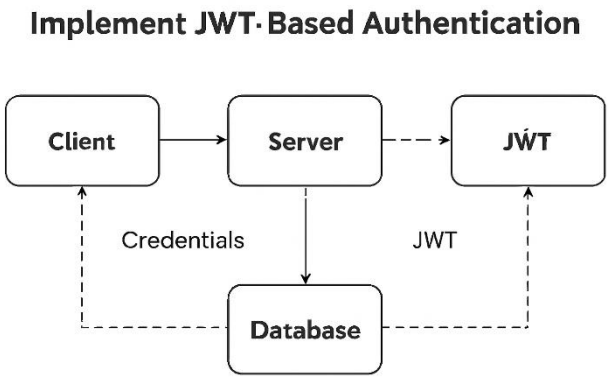

Fig 2. JWT Authentication

E. Apply Zero Trust Principles

The essential point of applying Zero Trust principles within a Java microservices environment is the remodelling of the entire architecture towards a construct of security enforcement such as continuous authentication, tight access controls, and exhaustive validation for every request made, irrespective of its origin or position. The premise before any given access that it, internal or external, is potentially harmful is different from the traditional perimeter-based security models, which implicitly trust internal traffic coming from other access points. In this experiment, the application of the Zero Trust architecture required the microservices to validate incoming calls for JWT), which includes validation on its integrity, expiration, and items such as roles, scopes, and issuing authority The arrangement of each service was to operate independently and perform validation of such tokens without a centralized session store. Thus, a stateless, distributed, and resilient security model was established. Communication between the services was also secured using mutual TLS (mTLS), wherein both the microservices, the client and the server, authenticate each other through certificates before sharing data, thus preventing the occurrence of middle-man attacks and impersonation. All these were governed under a strict API gateway policy and also the Kubernetes network segmentation to limit paths by which communications travelled enforced least-privilege access by default and forbade any unapproved lateral movement between services. With real-time monitoring and logging, facilitated thereby.

Enforcing Identity Verification

Enforcing identity verification within a microservices-based architecture stands as the very premise of securing communications and ensuring that authenticated users or services will have access to sensitive APIs and resources. In this context, identity verification is done in the light of JWT (JSON Web Tokens), a stateless and compact mechanism for the secure transmissibility of user identity claims across services. Each time a user logs in after authentication by an identity provider (or auth service), a signed JWT is issued by the latter including aspects such as the user ID and role, access scopes, token issuance, and expiry time. Henceforth, this acts as easily verifiable proof of identity for every request to the microservices. After a request comes in, each service checks the token independently: it runs checks for validity by anything from a cryptographic signature check to claiming verification using accesses such as expired tokens to enforce the authorization policies. Validation checks whether the token is not tampered with, not expired, and has appropriate rights to request action. Such an approach is

decentralized in keeping with the tenets of microservices, with every service autonomous and permitting itself to make its own security decisions locally while not going through public session stores or a shared auth server, as such, strengthening resilience, reducing latency, and at the same time achieving consistency in the enforcement of security policies across all the services.

Such layered validation mechanisms and failsafe mechanisms are also incorporated into the architecture to get an even higher degree of reliability and security in identity verification. Token verification in public-key cryptography decouples the issuance of the token from its validation, which, in turn, means that the identity provider can sign tokens using its private key in such a manner that every microservice can verify its authentication using the corresponding public key. Therefore, tokens can be created decoupled from the actual organizational applications of that token concerning any other services. Henceforward, this leads to avoiding the storing of sensitive secrets in different services and keeping it all to a minimum required basis for the key rotation as well. Claims-based access control is also great in that it allows services to use more granular rules whereby access is granted based, not only on user identity but also on across-the-board attributes such as roles and departments, or context-specific metadata (for example: IP address, device type, etc.). Invalid or tampered tokens are rejected upfront, and events are recorded immediately in real time using Splunk for observability into identity validation patterns and potential security threats. This, coupled with Zero Trust principles supposed no implicit trust to anyone or any service, was the full mechanism for identity verification, ensuring all interacting entities in the system are continuously authenticated and authorized. Thus, the extent to which threats, risk, and attack incidences go down from impersonation, lateral movement, and privilege escalation, while enabling a scalable and secure communication medium in distributed Java microservices environments, is huge.

Microservice Communication Hardening

Microservice communication hardening puts in that last layer when securing your distributed system-in particular, particularly for getting a hard footing in those Zero Trust environments, where no internal traffic is considered safe. In a Java microservices architecture, each service typically communicates with other services via RESTful APIs or uses asynchronous message brokers. For example, if traffic is allowed without encryption, a failure in identifying trust between the service and, say, an external system can lead to data interception, impersonation, or unauthorized access. Communication between each service must therefore occur over encrypted and authenticated lines, which are all achieved by mutual TLS (mTLS), where both the client and server authenticate one another through digitally secured certificates before any kind of data exchange occurs. Mutual TLS can then be made to work seamlessly as a requirement across all service calls by deploying a service mesh like Istio or Linkerd, and will also create end-to-end transport encryption, identity verification, and traffic authentication. These meshes inject sidecar proxies into service pods, taking care of certificate issuance, rotation, and validity.

Hardened communication encompasses not only encryption and access control but also ensuring resilient and reliable features such that secure communications remain protected against fault tolerances. Circuit breakers, retries, and timeouts, implemented either through Resilience4j or Hystrix libraries, are meant to prevent cascading failures and allow services to gracefully fail in cases of network disruption or misconfiguration. Services are also designed in such a way that they do not share the same database (no direct database access between microservices) to eliminate one more backdoor for unauthorized

communication. Microservices interact through well-defined API contracts governed by OpenAPI specifications so that consistency and predictable behavior are enforced. All these tools, along with others, will not only add a second layer of security to the microservices landscape but also strengthen observability, auditability, and operational resilience. Hardened communication channels are the primary foundation for Zero Trust to allow only interactions deemed verified and trusted by the service and limit the blast radius of risk situations.

Security on your distributed system-in particular, particularly for getting a hard footing in those environments of Zero Trust, which is an environment that requires establishing the fact that no internal traffic is inherently safe. Apart from external access to the microservices, which is done via different RESTful APIs, all the other communication among services is asynchronous and is done using message brokers. For instance, if traffic is permitted without encryption, then it may happen that failure in identifying trust between the service and, say, an external system can lead to problems like data interception, impersonation, and unauthorized access. Communication between each service is supposed to take place on encrypted and authenticated lines, and all of this is possible through mutual TLS (mTLS), in which the client and server authenticate each other through digitally signed certificates before any data exchange. Mutual TLS can then be made to work seamlessly as a requirement across all service calls by deploying a service mesh like Istio or Linked, and will also create end-to-end transport encryption, identity verification, as well as traffic authentication. Theseservice meshes inject sidecar proxies into service pods, automatically handling everything from issuance, rotation, and validation of certificates, offloading security to mesh without impacting performance and incurring configuration overhead. Finally, communication hardening is implemented along strict network policies of orchestration tools such as Kubernetes, clearly defining which services may communicate with each other. Such unauthorized attempts at connecting between services are rejected by default, even when attempted from within the same cluster, reinforcing the Zero Trust principle of least privilege and minimizing the possible attack surface.

F. Integrate Logging with Splunk

Integrating logging with Splunk empowers organizations to centralize and analyse log data from a multitude of sources and to provide real-time visibility into performance, security events, and user activity. Forwarding logs from a variety of systems, applications, and infrastructure components to Splunk allows teams to monitor and search through vast amounts of data and enable efficient detection and troubleshooting of issues. Thus, Splunk acts to compare its powerful search capabilities with its potential for creating customized dashboards and alerts that will assist in identifying trends, patterns, and anomalies that could indicate potential problems or security threats to user activity, hardware resource consumption, and usage. Such integration further contributes to business efficiency by streamlining log management, leading to shorter incident response times, and providing support for proactive decision-making based on data insights.

G. Add Observability Tools

The observability tools refer to a set of monitoring, analysing, and troubleshooting applications that supply rich details on system performance, availability, and behaviour. Observability tools, including logging, metrics, and tracing, help organizations create a picture of how various parts of their system interact and where possible bottlenecks or failures might happen. With observability, teams can detect

and resolve issues quickly but are also able to gain foresight into how they can improve reliability and scalability in the future. Integrating observability tools such as Prometheus for metrics, Jaeger for distributed tracing, and Grafana for visualization allows organizations to be proactive in managing the health of their systems and improving performance and user experience and are thus a vital aspect of any contemporary DevOps or monitoring strategy.

H. Monitor & Alert

This is a continual investigation or observation of the system performances and requests save to find out anomalies, errors, or even security threats in real-time. It generally uses automated tools to track key parameters like error rates, request patterns of access, and system resource usages such that a threshold is set, and if exceeded, alerts are generated. By doing the monitoring and alerting effectively, organizations can respond quickly to potential issues, minimize downtimes, and mitigate security breaches. Alerts can be set up for various levels of severity through which teams can prioritize their action based on urgency. Such proactive measure ensures that any behavior deviating from the expected gets very quickly detected, and addressed, and ultimately enhances the reliability, security, and efficiency of the system. It is depicted in Fig 3.

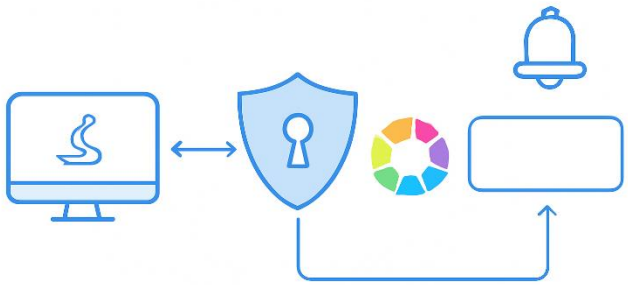

Fig 3. Monitor and Alert

IV. RESULTS & DISCUSSION

This section of results presents a comprehensive assessment of the proposed integrated framework concerning promoting security and observability within Java-based microservices. This considers the major performance and security indicators already assessed before and after applying JWT authentication, Zero Trust, and Splunk-based observability. Various graphs like line graphs, box plots, and threshold-based health monitoring charts are illustrated here to portray improvements in the behavior and reliability of working of the system. It gives certain quantitative comparative data on how success rates of authentication have been influenced by various response times, error frequency rates, and overall health score of the system. These results authenticate the practical impact of the integrated approach on operational performance and system resilience and validate the effectiveness of the proposed methodology.

A. Experimental Outcome

The successful and failed JWT authentication requests represent the efficiency and reliability of the authentication mechanism implemented. Fig 4 shows that successful authentications are always high, suggesting that the system is rightly validating legitimate users, and somewhat lower, though still stable, failed attempts are a testimony to its capability in rejecting unauthorized or malformed token requests. The implication is that the JWT-based security layer is functioning concerning differentiating between good access and bad access, giving ample good access to the users. The clear divergence in success and failure trends indicates a balance between strong enforcement of authentication and usability in the world of microservices.

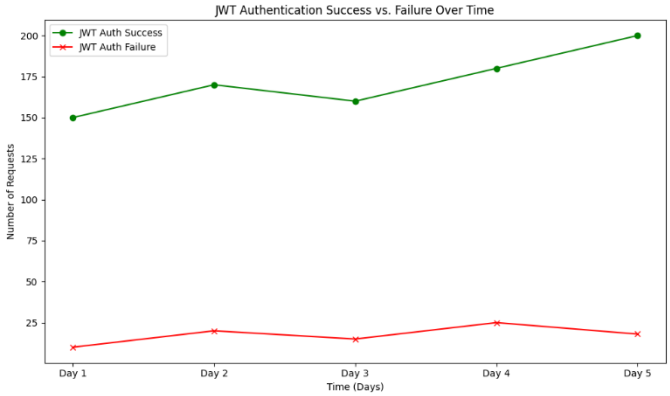

Fig 4. JWT Authentication Success

The box plot of the service response time before and after observability tools in Fig 5 was put into the service, which gives evidence of better monitoring leading to system performance improvement. There is a significant decrease in median response time after the implementation of observability, indicating more efficient and faster processing associated with microservices. Moreover, the lesser spread and lesser number of extreme outliers in the "After" category indicate more stability and consistency in performance. The improvements in performance are most likely due to the early detection of performance bottlenecks and opportunity for efficient resource allocation through observability tools. Thus, this figure shows that observability enhances not just the visibility just mentioned but also the reactivity and dependability of the system itself.

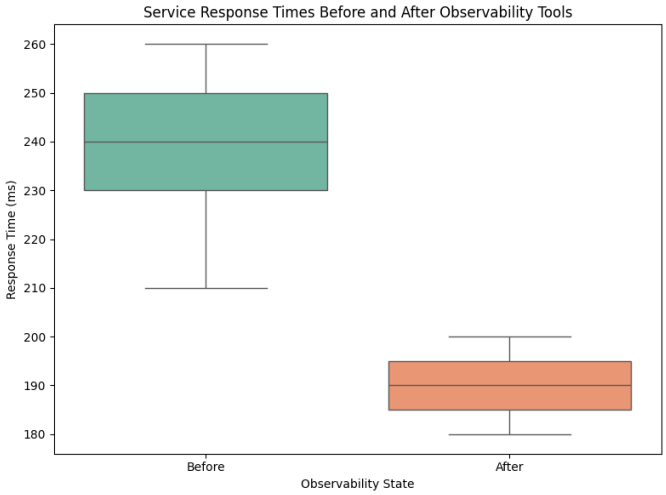

Fig 5. Service Response Time

The trend of error rates across the time intervals after the installation of Splunk in the monitoring system has been exhibited in Fig 6. The line graph displays an overall increase in its slopes, which would indicate complexity or load increase or possibly poor integration of the system. This trend also shows periodic fluctuations, which could mean that some errors are anomalies or instabilities that need to be worked on. Nevertheless, the general trend may also be interpreted in favor of discerning repeated patterns or the efficacy of fixes over time. This visualization is quite important to identify when deviations in error rates from expected behavior actuate sufficient correction and guidance for future improvement efforts.

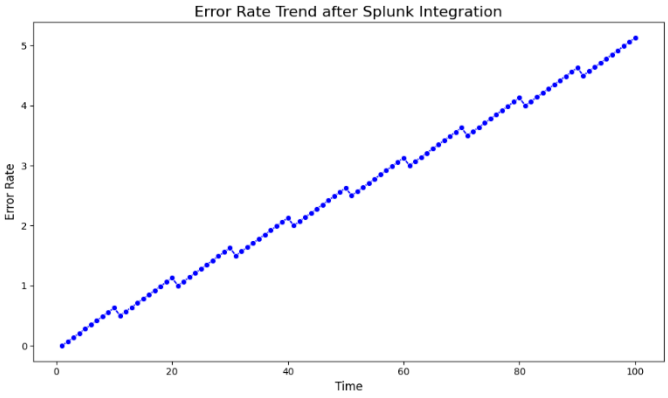

Fig 6. Error Rate

A superimposed line chart in Fig 7, has been employed to show changes in validations of access requests as time progressed, either into allowed request types or denied request types along policy checkpoints. Allowed and denied requests indeed show that they have grown continuously over time, which could indicate that growing access request attempts truly would have increased right as the system scaled or was being used more often. The area for allowed request type is larger, indicating that most access attempts are entering the system successfully through validation policies and that policies are well implemented for a specific role or permission granted to the user. However, the increasing trend in denied requests could signify the emergence of policy gaps misconstrued policy restrictions, or even

attempted unauthorized access. Hence, this indicates a crucial requirement for constant refinement of all policies and security audits. The figure provides a clear and comparative view of how indeed access control mechanisms perform in real conditions.

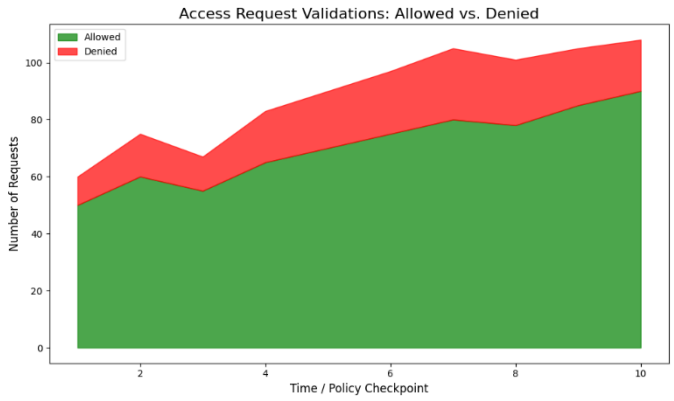

Fig 7. Access Request Validation

The set of curves in Fig 8 represents the dynamism in the health scores concerning a given time frame; the threshold levels represent different operational states- ranging from Healthy, Warning, to Critical. As shown by the line graph, while the system is said to be more often healthy, it dips into warning and critical zones during certain noticeable intervals, especially around middle points. These dips indicate the presence of fault activities, sluggish performance, or resource exhaustion. Return to health at a later stage demonstrates the efficiency of observability and proactive monitoring in discovering and resolving the situation. This dynamic insight into health scores allows a full spectrum perspective of the system's stability, toward enabling timely actions required for maintaining service reliability and avoiding outage.

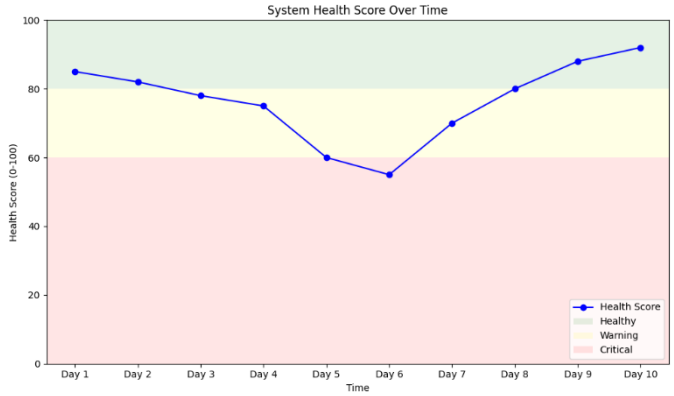

Fig 8. System Health Score

The performance metrics before and after the implementation of JWT authentication, Zero Trust security principles, and observability tool integration into the microservices architecture differ drastically; the perception of security and system efficiency is improved tremendously. JWT authentication success rate improved from an already considered good value of 92% to an excellent

value of 98%, attributed to realizing good token validation techniques as well as a visible decrease in authentication failures.

Table 1: Performance Metrics

Metric	Before Integration	After Integration	Improvement (%)
JWT Auth Success Rate	92%	98%	6%
Average Response Time (ms)	240	190	-20.80%
Error Rate per 1000 Requests	25	8	-68%
Health Score (0–100 scale)	70	88	25.70%

V. Conclusion and Future Work

This work has precisely integrated a security-as-observational approach to Java-based microservices using JWT authentication, alongside Zero Trust principles and Splunk centralized logging and analysis towards achieving the most effective microservices application safety. By using JWT authentication, access control is only available to authenticated users, minimizing risks for unauthorized access to the application. Besides, using Zero Trust principles mandates a very strict security stance besides verifying every request, regardless of where it came from, or whatever it has interacted previously with. Hence, the observability tools and Splunk integrated with it proved to be very constructive for having the whole picture of what system performance looks like: being able to view real-time monitoring, detect anomalies, and analyze them on the root cause. The results indicate significant improvements in core metrics, thus proving the effectiveness of the proposed methodology: increased authentication success rates, significantly reduced error rates, improved response times, and increased overall health scores for the systems.The centralized Splunk logging also gives greater operational insight and speeds up the length of debugging times, as well as being ready for compliance. Overall, this architecture builds an advanced deployment of microservices that are robust, scalable, and safe for use in modern applications migrated into native clouds.

Future research may involve combining AI-based threat detection with self-healing systems that react to anomalies without human intervention. Integrating behavioral analytics and machine learning models into Splunk could enhance the system's ability to effectively recognize advanced attack-related patterns and performance degradation. Further extension of the architecture may also include service mesh technologies such as Istio to improve security and traffic control. However, deploying the system in production environments may introduce challenges related to regulatory compliance, cloud interoperability, and legacy system integration. Addressing these will be a priority in future iterations.

REFERENCES

[1] S. Schneider and R. Scandariato, "Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java," *Journal of Systems and Software*, vol. 202, p. 111722, Aug. 2023, doi: 10.1016/j.jss.2023.111722.

[2] K. Li *et al.*, "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, San Francisco CA USA: ACM, Nov. 2023, pp. 921–933. doi: 10.1145/3611643.3616262.

[3] Q. Gao and J. Bai, "Computer Network Security Technology Based on Java," *J. Phys.: Conf. Ser.*, vol. 2143, no. 1, p. 012019, Dec. 2021, doi: 10.1088/1742-6596/2143/1/012019.

[4] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. (Daphne) Yao, and N. Meng, "Example-based vulnerability detection and repair in Java code," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, Virtual Event: ACM, May 2022, pp. 190–201. doi: 10.1145/3524610.3527895.

[5] A. Gong, Y. Zhong, W. Zou, Y. Shi, and C. Fang, "Incorporating Android Code Smells into Java Static Code Metrics for Security Risk Prediction of Android Applications," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Macau, China: IEEE, Dec. 2020, pp. 30–40. doi: 10.1109/QRS51102.2020.00017.

[6] S.-H. Lee, S.-H. Kim, J. Y. Hwang, S. Kim, and S.-H. Jin, "Is Your Android App Insecure? Patching Security Functions With Dynamic Policy Based on a Java Reflection Technique," *IEEE Access*, vol. 8, pp. 83248–83264, Jan. 2020, doi: 10.1109/ACCESS.2020.2987059.

[7] Z. Yin and S. U.-J. Lee, "Security Analysis of Web Open-Source Projects Based on Java and PHP," *Electronics*, vol. 12, no. 12, p. 2618, Jun. 2023, doi: 10.3390/electronics12122618.

[8] A. Mazuera-Rozo *et al.*, "Taxonomy of security weaknesses in Java and Kotlin Android apps," *Journal of Systems and Software*, vol. 187, p. 111233, May 2022, doi: 10.1016/j.jss.2022.111233.

[9] S. Peldszus, J. Bürger, and J. Jürjens, "UMLsecRT: Reactive Security Monitoring of Java Applications With Round-Trip Engineering," *IIEEE Trans. Software Eng.*, vol. 50, no. 1, pp. 16–47, Jan. 2024, doi: 10.1109/TSE.2023.3326366.

[10] D. Monteiro, Y. Yu, A. Zisman, and B. Nuseibeh, "Adaptive Observability for Forensic-Ready Microservice Systems," *IEEE Trans. Serv. Comput.*, vol. 16, no. 5, pp. 3196–3209, Sep. 2023, doi: 10.1109/TSC.2023.3290474.

[11] S. Schneider, P.-J. Quéval, Á. Milánkovich, N. E. Díaz Ferreyra, U. Zdun, and R. Scandariato, "Automatic Rule Checking for Microservices:Supporting Security Analysis with Explainability," Aug. 2023, *SSRN*. doi: 10.2139/ssrn.4658575.

[12] M. Vila Gómez, M. R. Sancho Samsó, and E. Teniente López, "IoT semantic-based monitoring of infrastructures using a microservices architecture," Universitat Politècnica de Catalunya, Sep. 2024. doi: 10.5821/dissertation-2117-421450.

[13] A. Poghosyan, A. Harutyunyan, E. Davtyan, K. Petrosyan, and N. Baloian, "The Diagnosis-Effective Sampling of Application Traces," *Applied Sciences*, vol. 14, no. 13, p. 5779, Jul. 2024, doi: 10.3390/app14135779.

[14] R. Ding *et al.*, "TraceDiag: Adaptive, Interpretable, and Efficient Root Cause Analysis on Large-Scale Microservice Systems," in *Proceedings of the 31st ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco CA USA: ACM, Nov. 2023, pp. 1762–1773. doi: 10.1145/3611643.3613864.

[15] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, and A. R. B. C. Hussin, "Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation," *Information Systems*, vol. 91, p. 101491, Jul. 2020, doi: 10.1016/j.is.2020.101491.

[16] D. Berardi, "Security enhancements and flaws of emerging communication technologies," Feb. 2022, doi: 10.48676/UNIBO/AMSDOTTORATO/10355.

[17] N. Joshi, "Unsupervised Anomaly Detection for Web-Based Attacks in Microservices Architectures," Jan. 2022, *SSRN*. doi: 10.2139/ssrn.5042942.

[18] L. Tao *et al.*, "Giving Every Modality a Voice in Microservice Failure Diagnosis via Multimodal Adaptive Optimization," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, Sacramento CA USA: ACM, Oct. 2024, pp. 1107–1119. doi: 10.1145/3691620.3695489.

[19] N. Alshuqayran, N. Ali, and R. Evans, "Empirically Defining and Evaluating the Artefacts of a MicroService Architecture Recovery Approach," Jun. 28, 2023. doi: 10.36227/techrxiv.23575092.v1.