International Transactions on Electrical Engineering and Computer Science
Sairamakrishna et.al., Vol: 4, No: 1, pp: 01-12, March 2025

Enhancing Cloud-Native Applications: A Comparative Study of Java-To-Go Micro Services Migration

Sairamakrishna BuchiReddy Karri[1], Chandra Mouli Penugonda[2], Srujana Karanam[3], Mohd Tajammul[4], Srinivasarao Rayankula[5], Prasad Vankadara[6]

Abstract: Moving microservices from Java to Go creates great opportunities for performance, scalability, and resource efficiency. Nonetheless, such a move comes with other challenges related to infrastructure changes, deployment strategies, observability, and security. This paper tries to look at elements of paramount importance as concerned with Java-to-Go migration, thereby, interrogating the key hosting environments, containerization, and orchestration. Go as a light engine introduces one of the most cost-effective deployments as organizations lean towards cloud-native architectures and Kubernetes-based orchestration [24]. The transition likewise demands adaptation of the observability practices since Go applications utilize different tools compared to Java applications. Security topics currently include dependency management, API protection, and vulnerability scanning which are highly pertinent in keeping the application intact. However, with proper planning, these challenges can leverage the advantages Go provides, ultimately presenting it as an attractive option for microservices development. Future studies should look into automated migration tools, the process of standardized best practices, and refinements to security to ease this transition.

Keywords: Microservices Migration, Java to Go Transition, Performance Optimization, Scalability and Efficiency, Containerization and Deployment.

History
Received: 25-11-2024; Revised: 18-01-2025;
Accepted: 25-01-2025

Sairamakrishna BuchiReddy Karri
sairamakrishna.karri@gmail.com

[1]Independent researcher, Sr. Lead Software Developer, Solution Architect, Farmington, MI - 48335, USA
[2]Independent researcher, Sr. Lead Software Developer, Solution Architect, Celina, TX - 75009, USA
[3]Independent researcher, Sr. Data Engineer/ Sr. Software Developer, Dallas, TX – 75078, USA
[4]Department of CSA, Sharda University, Greater Noida - 201306, India
[5]Independent researcher, Technical Architect/ Sr. Data Architect, Mclean, Virginia - 22102, USA
[6]Independent researcher, Sr. Lead Software Developer, Mclean, Virginia - 22102, USA

1. Introduction

1.1 Background of Microservices Migration

Modern software development is on the verge of embracing micro-services architecture because of very many reasons including scalability, flexibility, and maintainability. With micro-services, firms are taking to breaking monolithic applications into smaller independent services that can be deployed and administered separately. Java has long been the mainstay programming language for micro-services with a robust ecosystem with numerous libraries and architecture independence. However, and now that performance optimization and efficiency have become hot topics of discussion, companies continue to look for alternatives such as Go, which-Golang can be a better case for the microservices infrastructure [1].

https://doi.org/10.62760/iteecs.4.1.2025.127

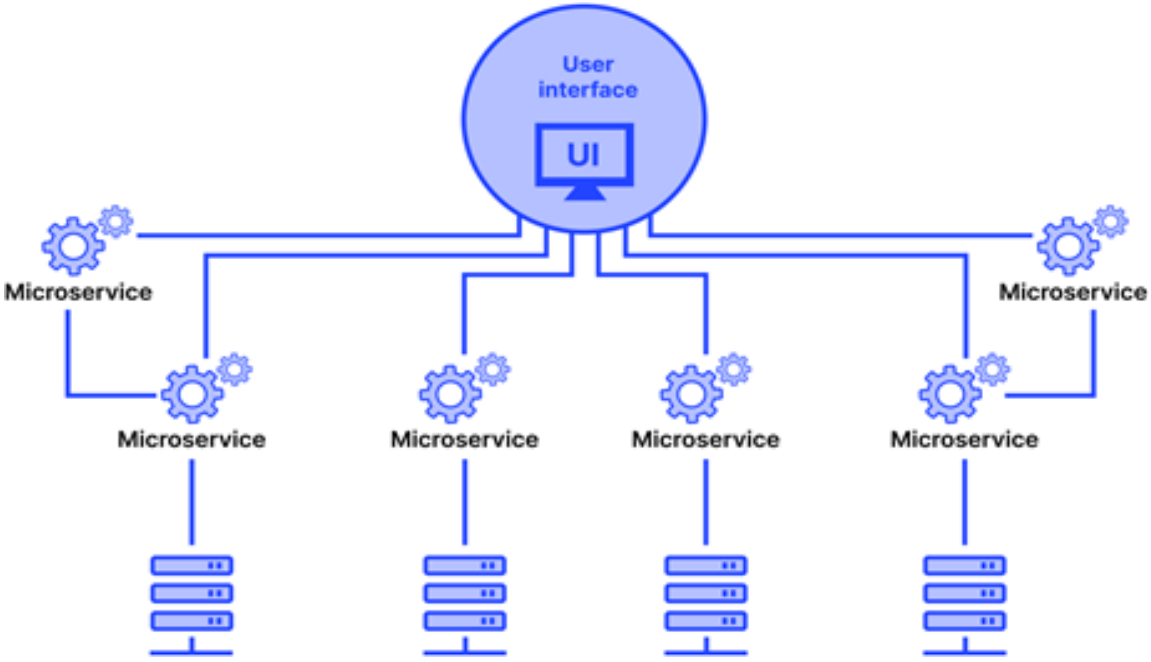

Fig. 1: Microservices Architecture Diagram [21]

Microservices architecture, as shown in Fig. 1, showcases decentralized approach. Where communicate with each other while being managed separately. This model aligns with the increasing adoption of microservices in modern software development, as it ensures enhanced scalability, flexibility, and maintainability. By breaking down monolithic applications into smaller components, organizations can streamline deployment, improve fault isolation, and optimize resource utilization. Java has traditionally been a preferred choice due to its rich ecosystem and platform independence. However, as performance and efficiency gain prominence, alternatives like Go (Golang) are emerging as strong contenders, offering lightweight concurrency and better resource management, making them a viable option for microservices infrastructure.

1.2 Reasons to migrate from Java to Go

Java still being a good choice for building micro-services, it comes with its own share of pain points-higher memory consumption, longer start-up times, and overheads arising from garbage collection. Go, on the other hand, compiles statically and that brings some performance, high concurrency, and low resource usage into the fold. Factors for the need to migrate from Java to Go are,

- ***Performance Improvements:*** Go systems perform better than those implemented in Java [2].
- ***Concurrency Management:*** Go's go-routines are lightweight alternative threads, thus making it a better option than Java's thread-based model for high- performing applications
- ***Much Lower Resource Usage:*** Being compiled and having a smaller memory footprint, Go leads to lower infrastructure costs with better scaling capabilities [3].
- ***Eased Deployment:*** Any Go program builds to a single static binary with no dependencies, making deployment easier and faster

Fig. 2 presents an in-depth comparison of programming languages based on key factors. These factors include popularity, which reflects the adoption rate among developers, and cross-platform development, which determines the ease of running applications on multiple operating systems. Performance plays a crucial role in execution speed, while usage highlights the practical implementation of the language across industries. The comparison further considers frameworks that support development and memory management for optimizing resource utilization.

Fig. 2: Key Factors in Programming Language Comparison [22]

Additionally, aspects such as concurrency for handling parallel processing, pointers and reflection for advanced memory operations, and code examples for clarity are evaluated. The syntax and ease of coding are examined to determine learning curves, alongside community support, which impacts troubleshooting and development. Lastly, error handling capabilities are assessed for their role in debugging and stability. This structured comparison helps developers choose a language that aligns with their project requirements.

1.3 Industry Trends and Adoption

Moving applications from Java to Go is an ever-growing trend in recent times, particularly in cloud native environments and high-performance computing. Technology giants, such as Google, Uber, and Netflix, have adopted Go on account of the latter's fast speed of performance, efficiency, and simplified deployment. Newer reports from the industry express that Go's use is higher now than before, especially in the categories of DevOps, cloud computing, and backend services [4]. These survey results further show that the switch to Go has vastly improved system performance, developer productivity, and operational efficiency. The increasing containerized environment adoption and Kubernetes give further momentum to this shift toward Go-based microservices.

1.4 Research Problem and Significance

Moving applications from Java to Go is an ever-growing trend in recent times, particularly in cloud native environments and high-performance computing. Technology giants, such as Google, Uber, and Netflix, have adopted Go on account of the latter's fast speed of performance, efficiency, and simplified deployment. Newer reports from the industry express that Go's use is higher now than before, especially in the categories of DevOps, cloud computing, and backend services [5]. These survey results further show that the switch to Go has vastly improved system performance, developer productivity, and operational efficiency. The increasing containerized environment adoption and Kubernetes give further momentum to this shift toward Go-based microservices.

1.5 Objectives of the Review

The main objectives of this systematic literature review (SLR) are,

- To analyze the motivations behind migrating Java microservices to Go.
- To evaluate the performance benefits and challenges associated with migration.
- To identify best practices and methodologies for a smooth transition.
- To explore industry case studies and real-world implementations of migration.
- To provide recommendations organizations migration considering for Java-to-Go.

This review aims to provide a comprehensive understanding of Java-to-Go microservices migration, facilitating informed decision-making for organizations seeking to optimize their software architecture.

The remaining paper is organized into various sections. Section 2 provides the methodology, Section 3 presents about the key factors to be considered during the migration, Section 4 about the performance, stability considerations, Section 5 about the infrastructure, deployment considerations and lastly Section 6 talks about the conclusions.

2. Research Methodology

There will be a systematic literature review methodology used in this study for collecting. analyzing, and synthesizing existing research on the migration of Java microservices to Go [6]. SLR is a well-defined and reproducible process to identify the relevant literature, the corresponding extraction of findings, and provide summary-level qualitative insights

PRISMA Framework

The PRISMA framework consists of four key phases:

- ***Identification:*** Relevant studies have been identified via database searches through sources such as ACM Digital Library and Google Scholar.
- ***Screening:*** Primary search results go through other screening processes based on pre-defined inclusion and exclusion criteria to filter for irrelevancy of the studied cases.
- ***Eligibility:*** Selected studies are reviewed in detail to ensure the studies meet the aims of the research and provide the insights needed.
- ***Inclusion:*** The final selection of studies included for qualitative and quantitative synthesis represents the source of data for a literature.

The use of the PRISMA framework ensures that the research follows a systematic and reproducible approach, minimizing bias and enhancing the reliability of the findings.

2.1 Search Strategy and Databases Used

The search strategy involved querying multiple scholarly databases to ensure comprehensive coverage of relevant studies. The databases used include,

- ACM Digital Library
- Google Scholar

(Java microservices" OR "Java-based microservices") AND ("Go" OR "Golang") AND ("migration" OR "transition" OR "conversion" OR "porting") AND ("performance" OR "scalability" OR "efficiency" OR "latency" OR "resource utilization") AND ("challenges" OR "best practices" OR "case study" OR "adoption").

Table. 1: Results on dataset

Dataset	Search String	Result
Google Scholar	("Java microservices" OR "Java-based microservices") AND ("Go" OR "Golang") AND ("migration" OR "transition" OR "conversion" OR "porting") AND ("performance" OR "scalability" OR "efficiency" OR "latency" OR "resource utilization") AND ("challenges" OR "best practices" OR "case study" OR "adoption")	110
ACM Digital Library	('Java microservices' OR 'Java-based microservices') AND ('Go' OR 'Golang') AND ('migration' OR 'transition' OR 'conversion' OR 'porting') AND ('performance' OR 'scalability' OR 'efficiency' OR 'latency' OR 'resource utilization') AND ('challenges' OR 'best practices' OR 'case study' OR 'adoption')	9

2.1.1 Inclusion and Exclusion Criteria

Inclusion Criteria

- Published within the last five years Open-access publications
- Written in English
- Relevant to Java-to-Go migration Total Article
- N-119 09
- Filter 1
- Identify and Exclude books and Conference papers N-14
- Empirical studies, case studies, and systematic reviews

Exclusion Criteria

- Non-English publications Studies older than five years

- Studies not specifically discussing Java-to-Go migration
- Unavailable full-text articles

Table. 2: Results after applying inclusion & exclusion

Dataset	Search String	Result
Google Scholar	("Java microservices" OR "Java-based microservices") AND ("Go" OR "Golang") AND ("migration" OR "transition" OR "conversion" OR "porting") AND ("performance" OR "scalability" OR "efficiency" OR "latency" OR "resource utilization") AND ("challenges" OR "best practices" OR "case study" OR "adoption")	3
ACM Digital Library	("Java microservices" OR "Java-based microservices") AND ("Go" OR "Golang") AND ("migration" OR "transition" OR "conversion" OR "porting") AND ("performance" OR "scalability" OR "efficiency" OR "latency" OR "resource utilization") AND ("challenges" OR "best practices" OR "case study" OR "adoption")	1

Study Selection Process (PRISMA Framework)

The study selection followed the PRISMA framework, consisting of four phases:

- ***Identification:*** An initial search retrieved 110 results from Google Scholar and 9 results from ACM Digital Library.
- ***Screening:*** Studies were filtered based on the inclusion and exclusion criteria, reducing the results to 3 from Google Scholar and 1 from ACM.
- ***Eligibility:*** A full-text review was conducted to ensure relevance to Java-to-Go migration, retaining the most pertinent studies.
- ***Inclusion:*** The final selected studies were used for data extraction and synthesis.

2.2 Data Extraction and Synthesis Methods

This systematic and rigorous analysis of Java-to-Go migration proceeded by extracting data derived from the scrutiny of selected studies that shed light into diverse aspects of microservices architecture and migrations strategies. The studies addressed are as follows.

Article [1]: Transitions from monolithic applications to microservices, stresses the most significant challenges, including structural complexity, scattered application logic, and external framework dependencies. These findings are a baseline to establish which can lead to further discussions on the actual migration from Java- based monoliths to more modular architectures, such as Go-based microservices.

Article [2]: Studies the role of microservices in edge computing environments, emphasizing aspects that include orchestration, autoscaling, and deployment. These insights become relevant in assessing how Java and Go applications manage resource management, scalability, and performance in distributed systems.

Article [3]: A systematic literature review of testing in microservices, identifying gaps in validation approaches. In this way, this study aids in understanding the different approaches on how reliability, testing strategies, and resiliency in microservices-based environments contrast between Java and Go.

Article [4]: Explores machine learning-driven decomposition of monolithic applications into microservices by providing insights into automated migration strategies. Findings from this study now feeds the discussion on how to better optimize Java-to-Go migration using Al-driven approaches.

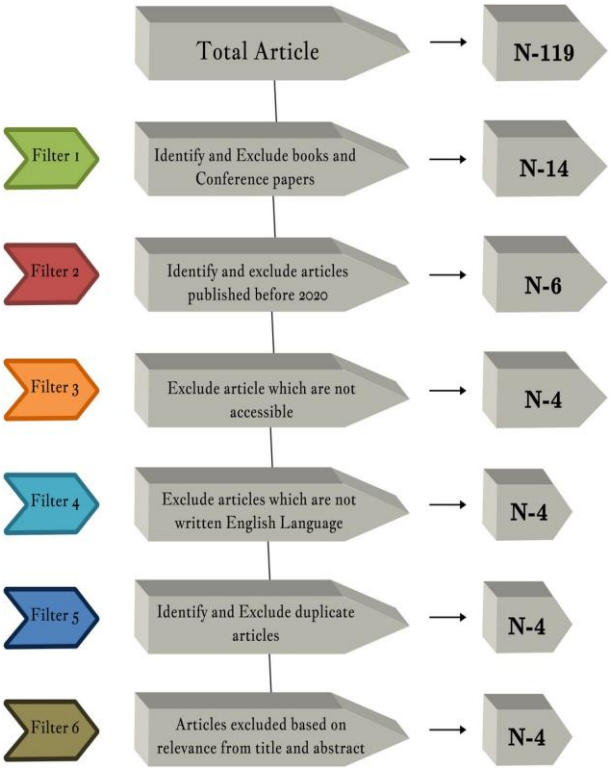

Fig. 3: Prisma Flowchart

3. Key Factors Driving Migration

Limitations of Java in Micro services Architecture

- High memory consumption because of the JVM (Java Virtual Machine).
- Added startup time prohibits fast scaling. Concurrency management makes it complex and a bottleneck possibility.
- Garbage collection overhead offsets high-load performance [7].
- The very verbose nature complicates the surface-level maintenance of development and maintenance

3.1 Benefits for Go for Microservices

Scalability:

- Allows lightweight goroutines and efficient concurrency management.
- Optimized for cloud-native applications with deployment in containers [8].

Resource Efficiency:

- Low memory footprint due to execution of compiled binaries.
- Efficient Garbage Collection with very low runtime overhead [9].

Simplicity:

Short syntax leads to lesser complexity in development and maintenance.
Built-in support for dependency management and testing tool.
Hosting & Deployment: Java applications often rely on heavyweight servers, while Go's self-contained binaries simplify cloud-native deployments [10].
Scalability: Java's thread-based concurrency 18 resource-intensive, whereas Go's lightweight goroutines enable efficient parallel processing [11].

3.2 Performance Comparison

Startup & Execution Speed: Java has slower startup times due to JVM initialization, while Go starts instantly, improving response times.
Concurrency: Java's thread model requires more memory, whereas Go's goroutines handle concurrency more efficiently.
Garbage Collection: Java's GC can cause performance delays, while Go's optimized GC ensures lower latency.

Go offers better efficiency, scalability, and speed, making it ideal for cloud-native applications, while Java remains strong for enterprise use cases. Organizations should choose based on their specific infrastructure and performance needs [12].

4. Performance and Scalability Considerations

Multiple research studies contrasting Java and Go have shown that Go is faster in normal execution when compared to Java, mainly because it is strongly compiled with very little run-time overhead. Java, on the other hand, does rely on the processing authorization of the Java Virtual Machine and implements a plethora of other software layers that tamper with running speed; yet, it has its compensatory scheme in the form of Just-In-Time compilation and adaptive optimizations. Although these enhancements may improve Java's run-time performance in the longer haul, Go really takes this even further due to its compilation strategy of machine code that results in faster start-up times and consistently lower response latencies. Go, then, is a great fit for software applications requiring speedy applications, such as in microservices or command-line tools and generally cloud-native solutions [13].

4.1 Resource Utilization Comparison

In terms of using system resources more effectively especially about memory and CPU burden, Go is significantly better than Java. The other plus for Go is that its garbage collector allows lower latencies and In terms of using system resources more effectively especially about memory and CPU burden, Go is significantly better than Java. The other plus for Go is that its garbage collector allows lower latencies and lets the code keep running smoothly even under heavy loads. The Java garbage collector, however, consumes heavy resources thanks to the extra memory brings into the JVM and more complex garbage collection mechanisms. In the shape of G1GC and ZGC, the pause time has been drastically reduced. It is significantly heavier on memory than Go. While few platform specifications dictate such overhead reduction in resource-constrained environments—like

edge computing, embedded systems, and containerized applications—these show how Go is so much more advantageous.

4.2 Concurrency, Parallel Processing Differences

The second big sector of Go is a shining example of simplicity and efficiency as far as concurrency provision is concerned. The Go concurrency is based on an inexpensive but resourceful practical application, the goroutines [14]. Meanwhile, conventional Java threads tend generally to occupy a larger memory footprint when operated and require more explicit construction and destructuring efforts; goroutines, on the other hand, allow for scheduling by the Go runtime, therefore allowing many thousand if not millions to be spawned with only the littlest amount of overhead. This makes Go an ideal choice for applications that require extensive parallel processing, such as network services, real-time data pipelines, and event-driven architectures. Java, in contrast, uses a thread-based model that, while powerful and flexible, comes with greater complexity and resource consumption, making it less efficient for handling high-throughput concurrent workloads.

4.3 Reliability and Stability in Large Systems

Go's design philosophy is firmly rooted and worked through, along the lines of simplicity, predictability, and reliability, fitting for large-scale distributed system where it works. With minimal approach, runtime errors and unexpected behavior are much less likely to occur. Static typing and a small degree of syntax improve maintainability in complex applications. Further, by the absence of a virtual machine, unintended dependencies causing instabilities in production environments are minimized [15]. Therefore, Java, with its mature ecosystem and exceedingly large libraries, remains a great choice for enterprise applications, but sometimes brings in operational challenges conducting distributed and cloud-native deployments with its runtime dependencies and memory-heavy nature. Organizations focusing on future-proof, scalable, and ease-of- maintenance often select Go to develop high- performance backend services and microservices architectures.

5. Infrastructure and Deployment Considerations

5.1 Changes in Hosting and Infrastructure

When migrating from Java to Go, organizations may need to reconsider some of the aspects of their hosting environments. Go apps are naturally lighter and do not depend on the JVM; as a result, they are cheaper and easier to deploy. Key considerations with respect to hosting.

Streamlined resource consumption

Minimal consumption of resources permits companies to move away from heavily Java-inclined servers to light-weight virtual hosts, either in the form of cloud- native platforms or light-weight VMs [16].

Transitioning to Cloud-Native Architectures

Migrating to Go are interested in serverless execution environments like AWS Lambda, Google Cloud Run, or Azure Functions, which complement Go's faster starting times.

Reducing operational costs by using the cloud

Because they don't have to tune the JVM or control high-memory usage, these cloud operations will easily incur high-performance scalability early on due to low infrastructure costs.

Easier delivery pipelines

Because Go compiles to a single static binary, delivery becomes easier because there are none of the constraints and complexities of runtime dependencies and setups commonly associated with JVM-based applications.

5.2 Containerization and Orchestration Technologies

Go is naturally suited for containerized deployments, offering benefits such as image size, startup speed, and runtime efficiency. Organizations moving from Java to Go should consider.

Docker and Leaner Container Images:

Unlike Java applications that require huge base images primarily due to JVM dependencies, Go applications result in much smaller container images, reducing storage and network overhead [17].

Integration with Kubernetes

Many organizations use Kubernetes for orchestrating microservices, and Go's efficient resource consumption makes it a natural fit. Kubernetes roles include the management of scaling, load balancing, and service discovery for Go-based applications.

Stateless and Immutable Deployments

Go's statically compiled binaries allow for immutable infrastructure practices, making rolling out updates less of a concern with regard to runtime dependencies.

A More Efficient CI/CD Funnel

By not needing a runtime environment such as the JVM, Go allows CI/CD pipelines to become streamlined through faster producing as well as deployment.

5.3 Monitoring and Observability Challenges

If migrating from Java to Go, monitoring and observability become issues, as traditional Java monitoring tools may not directly apply. Organizations will adjust their monitoring stack to suit Go's ecosystem:

Tolling Differences: Java usually employs JMX or uses New Relic. It may also use Java-based Prometheus exporters [18]. Go applications do need to use entirely different monitoring alternatives such as:

- ***Prometheus:*** Commonly used for metrics collection and monitoring in Go applications.
- ***Open Telemetry:*** Provides distributed tracing support for observability in microservices.
- ***Jaeger:*** Helps to trace distributed transactions in Go applications.
- ***Grafana:*** Frequently used in the context of Prometheus for visualization of performance data.

Adjustments of Logging: In Java applications, the libraries usually belong to the Log4j or SLF4J families. Go applications across the board depend on other frameworks, like structured logging in either Logrus or Zerolog [19].

Performance Profiling: Tools like pprof are important for profiling Go applications as a substitute for profiling tools used with Java, like VisualVM.

5.4 Security Aspects of Migration

Security still stands as a priority in this transition from Java to Go, given the eco-systems plus coding practices that go hand-in-hand with the transition have also opened up new vulnerabilities. Issues of concern in the area of security could include the following:

Dependency Management: Dependencies can never be ignored in software development and Go does not use anything as elaborate as Java has with its wonderful package management possibilities (such as Maven or Gradle), instead Go's modules are used, which really requires tighter management and prioritization of software versions to avoid conflicts and vulnerabilities to develop [20-22].

API Security and Authentication: Transitioning to Go could also require the reconsideration of API security and migration of the authentication solution from Spring Security-based frameworks to Go OAuth2-based authentication, JWT, middleware security models, and others.

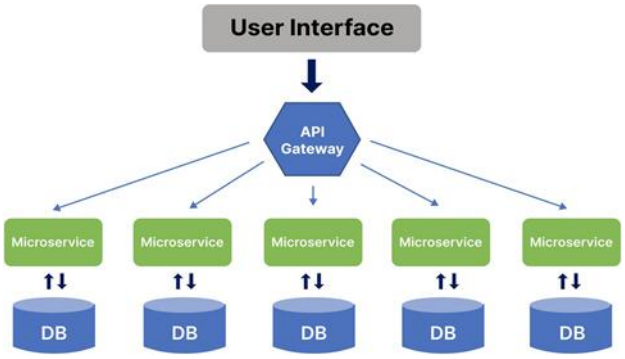

Fig. 4: API Gateway in Microservices Architecture [23]

Microservices architecture, where it acts as an intermediary between the user interface and various microservices. The API Gateway manages request routing, authentication, and security enforcement while ensuring efficient communication between services. Each microservice interacts with its dedicated database, maintaining data consistency and isolation. This structure enhances scalability, security, and performance by centralizing authentication mechanisms such as OAuth2, JWT, and middleware-based security models. As organizations transition

from traditional Spring Security frameworks to Go-based authentication solutions, implementing an API Gateway becomes crucial in maintaining secure and streamlined interactions across microservices.

Through the proper consideration of these infrastructure, monitoring, and security-related challenges as well as to the integration business, organizations guarantee seamless and secure transition from Java to Go along with the benefits of lightweight and efficient architecture permitted by Go.

6. Challenges and Open Research Areas

There are a variety of challenges that Java-to-Go migration companies consistently encounter during the transition process that cover the catalog of technical, organizational, and operational issues.

- ***Complexity of the Codebase:*** Large Java applications tend to have complex architectures featuring multiple dependencies and hence have set of considerable constraints on straightforward migration to Go [24]. The discrepancies in language paradigms by Java's object-oriented approach in contrast with Go's structural typing require massive restructuring efforts on the parts of Java-to-Go migrating teams [20].
- ***Lack of Expertise in Go:*** Many development teams do have great experience in Java, but not much in Go. This knowledge gap impacts not just the pace of migration, but can make systems debugging time-consuming and can lead to inefficient practices in development.
- ***Risks to Stability and Reliability of the System:*** Migrating from Java to Go takes considerable risk with respect to system reliability, especially when one transfers critical business applications. Issues of unoptimized garbage collection tuning, differences in exception handling, and concurrency model redesigns can greatly affect overall system performance [7].
- ***Tooling and Other Ecosystem Differences:*** Java has a mature ecosystem with extensive frameworks, libraries, and enterprise-grade tools. Despite a growing ecosystem, Go lacks direct alternatives to some Java-based tooling, forcing developers to find a new approach or write integrations themselves.
- ***Third-Party Dependencies and Compatibility:*** Many Java applications come heavily dependent on third-party libraries and frameworks; some of them might not even have direct alternatives in Go. Organizations should often rewrite similar functionalities from scratch or integrate different libraries whose level of support or maturity might not be ideal [11].
- ***Testing and Validation:*** To ensure that the migrated system demonstrates functional equivalence with the original Java application, validation testing involves significant effort. The dissimilarities in testing frameworks, mocking approaches, and logging mechanisms between Java and Go makes this complicated.

6.1 Limitations of Existing Migration Strategies

There are numerous strategies for migration. However, most of them are not standardized and extensively supported by guidelines, leaving considerable space for inconsistency in the execution and delivery of results [14].

- ***Performance Optimization:*** While Go is generally able to achieve better runtime performance, if tuned for good memory management, garbage collection settings, and concurrency mechanisms, it is up to the organization that accepts the project to recreate success with Go's true advantages.
- ***Debugging and Troubleshooting Issues:*** Java developers see many things wrong with Go's debugging feature. Of all the sophisticated profiling tools, possibly only the built-in tools supported with Go can assist well enough in helping a programmer go about coding [15].
- ***Refactoring Overheads:*** Converting Java code into Go often involves steep refactoring to settle differences in type systems, error handling, and concurrency models. This can prove to be time- consuming, and should not be not handled properly, can lead to unfortunate bugs.
- ***Data Migration and Integration Complexities:*** A good number of Java code applications interact with databases and message queues; likewise, there are APIs that really may not be either fully integrated or able to be adopted with Go-based solutions. It's still a challenge to

migrate the entire functional data set such that it is consistent and secure [12].

- ***Incremental Solutions versus Full Solutions:*** Organizations need to find a way of choosing between incremental migration, which is migrating services one at a time, and full-blown migration, where a system migrates in one go. Each solution bears its own risk, cost, and downtime

6.2 Knowledge Gaps and Future Research Directions

For the issues stated above, future works should encourage research towards finding working development tools, methodologies and standard operating practices for Java-to-Go migration. Future research possibilities include:

- ***Building Automated Migration Tools:*** These AI- based technologies can help in code translation, dependency analysis, performance optimization, and eliminate the burden of manual effort through your efforts of compassion by mitigating error-risk chances. Consideration has to be granted to machine-learning-based tools for migrations as they may ease the migration work process [13].
- ***Defining Codified Best Practices:*** A defined framework describing the sequential migration processes, coding conventions, and architectural orientations would ensure softer transitions for organizations and minimize the disruption associated with them.
- ***Automated Migration Tools:*** Develop intelligent, AI-enhanced tools that may help in translating code and managing dependencies and optimizing performance.
- ***Best Practices and Standardization:*** Setting forth well-drafted migration methodologies and design patterns and standard guidelines by which organizations would experience an easier transition.
- ***Real-World Case Studies:*** Conducting an in-depth study of successful Java-to-Go migrations across multiple industries to observe best practices and common failure points.
- ***Improved Observability and Debugging Tools:*** Enhance logging, tracing, and performance monitoring efforts for Go-with-an-eye-toward-make-it-as-good-as-Java-ecosystem.
- ***Security Enhancements:*** Building upon Go microservices security frameworks and best practices, assisting in aligning effective compliance with industry standards.

By focusing on these research areas, the industry can make Java-to-Go migration more accessible and efficient, enabling organizations to reap the full benefits of Go's capabilities in modern cloud-native environments.

Conflict of interest

The authors declared 'No conflict of interest'.

References

[1] S. Daniel, J. Martin, P. Genovesi, V. Maris, D. A. Wardle, J. Aronson, F. Courchamp "Impacts of biological invasions: what's what and the way forward", *Trends in ecology & evolution*, Vol. 28, No. 1, pp. 58-66, 2013. https://doi.org/10.1016/j.tree.2012.07.013

[2] Md. D. Hossain, T. Sultana, S. Akhter, Md. I. Hossain, N. T. Thu, L. NT Huynh, G. W. Lee, and E. N. Huh "The role of microservice approach in edge computing: Opportunities, challenges, and research directions", *ICT Express*, Vol. 9, No. 6, pp. 1162-1182, 2023. https://doi.org/10.1016/j.icte.2023.06.006

[3] M. Panahande and J. Miller "A Systematic Review on Microservice Testing", *Research square*, 2023. https://doi.org/10.21203/rs.3.rs-3158138/v1

[4] S. Korn, Md. S. Hossain Chy, M. A. Rahman Arju, T. Cerny and P. Rivas "Using Static Analysis to Aid Monolith to Microservice System Transformation: Tuning Fuzzy c-Means in a VAE-Based GNN Approach", *In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pp. 43-53, 2024. https://doi.org/10.1145/3691621.3694933

[5] F. S. Shoumik, M. I. M. M. Talukder, A. I. Jami, N. W. Protik and M. M. Hoque, "Scalable micro-service based approach to FHIR server with golang and No-SQL," *2017 20th International*

Conference of Computer and Information Technology (ICCIT), Dhaka, Bangladesh, pp. 1-6, 2017. https://doi.org/10.1109/ICCITECHN.2017.8281846

[6] A. Sara and M. AlAraj "Java-Powered AI: Using Code to Implement Intelligent Systems–Systematic Review", *International Journal of Theory of Organization and Practice*, Vol. 4, No. 1, pp. 100-106, 2024. [CrossRef]

[7] V. Victor, and P. Flores "A survey on microservices architecture: Principles, patterns and migration challenges", *IEEE access*, Vol. 11, pp. 88339-88358, 2023. https://doi.org/10.1109/ACCESS.2023.3305687

[8] H. Alexis, and Y. Ridene "Migrating to microservices", *Microservices: Science and Engineering*, pp. 45-72, 2020. https://doi.org/10.1007/978-3-030-31646-4_3

[9] F. Kaihua, W. Zhang, Q. Chen, D. Zeng, and M. Guo "Adaptive resource efficient microservice deployment in cloud-edge continuum", *IEEE Transactions on Parallel and Distributed Systems* Vol. 33, No. 8, pp. 1825-1840, 2022. https://doi.org/10.1109/TPDS.2021.3128037

[10] K. Gopal, L. Xun, R. Hasha, S. Bakht Ahsan, T. Pfleiger, R. Sinha, A. Gupta et al. "Service fabric: a distributed platform for building microservices in the cloud." *In Proceedings of the thirteenth EuroSys conference*, pp. 1-15, 2018. https://doi.org/10.1145/3190508.3190546

[11] R. S. Adalberto, J. Rubin, I. Beschastnikh, and N. S. Rosa "Improving microservice-based applications with runtime placement adaptation", *Journal of Internet Services and Applications,* Vol. 10, pp. 1-30, 2019. https://doi.org/10.1186/s13174-019-0104-0

[12] C. Pascal, C. Herzeel and W. Verachtert "A comparison of three programming languages for a full-fledged next-generation sequencing tool", *BMC bioinformatics*, Vol. 20, pp. 1-10, 2019. https://doi.org/10.1186/s12859-019-2903-5

[13] D. Patrick, M. Morris, S. R. Brandt, N. Gupta, and H. Kaiser "Benchmarking the parallel 1d heat equation solver in chapel, charm++, c++, hpx, go, julia, python, rust, swift, and java", In *European Conference on Parallel Processing*, pp. 127-138. Cham: Springer Nature Switzerland, 2023. https://doi.org/10.1007/978-3-031-48803-0_11

[14] W. Yingying, H. Kadiyala and J. Rubin "Promises and challenges of microservices: an exploratory study", *Empirical Software Engineering* Vol. 26, No. 4, art. No. 63, 2021. https://doi.org/10.1007/s10664-020-09910-y

[15] Y. Sun "A Comparison between Java and Go for Microservice Development and Cloud Deployment", 2021. https://urn.fi/URN:NBN:fi:amk-202104195171

[16] Y. Y. Chen, K. -H. Hsu and A. W. Hou, "MAT: Automating Go monolithic applications transform into microservices through dependency analysis and AST", *International Conference on Applied System Innovation (ICASI)*, Chiba, Japan, pp. 133-135, 2023. https://doi.org/10.1109/ICASI57738.2023.10179517

[17] A. M. Zohaib "Creating a microservice generator for go-based microservices", 2022. [CrossRef]

[18] A. Nabiil, B. H. Makmur, R. W. Wijaya, A. A. Santoso Gunawan and I. S. Edbert "Performance Analysis on Web Development Programming Language (Javascript, Golang, PHP)", *2023 International Conference on Information Technology and Computing (ICITCOM)*, Yogyakarta, Indonesia, pp. 6-11, 2023. https://doi.org/10.1109/ICITCOM60176.2023.10442358

[19] S. Simon and R. Scandariato "Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java", *Journal of Systems and Software*, Vol. 202, pp. 111722, 2023. https://doi.org/10.1016/j.jss.2023.111722

[20] Labián, Antonio, Jesús D. García-Consuegra, and Manuel Ortega "Migration of legacy Java desktop applications to collaborative Web", *Iberoamerican Workshop on Human-Computer Interaction*, pp. 200-209, 2023. https://doi.org/10.1007/978-3-031-57982-0_16

[21] A. Pouya, and D. Staegemann "Application of microservices patterns to big data systems", *Journal of Big Data*, Vol. 10, No. 1, art. No. 56, 2023. https://doi.org/10.1186/s40537-023-00733-4

[22] S. Siddhant, P. Singh, J. Sain, V. Shrivastava and A. Pandey "Modern Backend Development Technologies: A Comparative Review and Case Study", *International Conference on Emerging*

Trends in Expert Applications & Security, pp. 139-151, 2024. https://doi.org/10.1007/978-981-97-3745-1_12

[23] T. Gomathi, and V. Vijayalakshmi "An AI-driven Approach for Tailoring Java Programming Instructions", *In 2025 3rd International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT)*, pp. 1479-1484, 2025. https://doi.org/10.1109/IDCIOT64235.2025.10914687

[24] K. S. Buchireddy, B. R. Kumar, A. N. Reddy, A. S. Kumar "Cross-Domain Expert in Designing AI-Driven Microservices", *International Journal of Novel Research and Development*, Vol. 9, No. 12, no.c820-c824, 2024. http://doi.one/10.1729/Journal.42790

***