World Journal of Economics and Business Research
Print ISSN: 2960-0073
Online ISSN: 2960-0081
DOI: https://doi.org/10.61784/wjebr3083

TECHNOLOGY SELECTION OF ENTERPRISE JAVA WEB FRAMEWORKS BASED ON EVOLUTIONARY PATH ANALYSIS

ZhengLin Wang
School of Information and Electrical Engineering, Hebei University of Engineering, Handan 056038, Hebei, China.

Abstract: In response to the rapid iteration of enterprise-level Java Web frameworks and the lack of systematic theoretical guidance for selection decisions, this study proposes an innovative selection methodology based on the analysis of technical evolution paths. The main innovations are as follows: First, it reconstructs the three-dimensional "scenario-driver-paradigm" analysis model for framework evolution, systematically revealing the internal logic and technological inflection points in the transition from monolithic to cloud-native architectures, going beyond traditional version-based reviews. Second, it proposes a quantitative selection framework based on the dual-track "steady-state-agile-state" adaptation, which for the first time comprehensively models core dimensions such as development efficiency, runtime performance (startup time, memory usage), and cloud-native compatibility, clarifying the Pareto boundaries of mainstream frameworks such as Spring Boot and Quarkus under different constraints. Third, it constructs a hybrid architecture decision-making model based on technical debt and transformation cost assessment, providing enterprises with path planning and risk assessment tools for the smooth evolution from legacy systems to cloud-native architectures. This study validates the effectiveness of the method through empirical case comparisons, offering systematic theoretical foundations and practical guidance for enterprises to make forward-looking, low-risk architectural decisions in complex technological ecosystems.
Keywords: Java Web framework; Technical evolution path; Architectural selection; Innovative methodology; Cloud-native

1 INTRODUCTION

With the widespread adoption of cloud computing, containerization, and microservices architecture, enterprise application development is undergoing a paradigm shift from "virtual machine-centric" to "cloud-native-centric" [1]. Java, as a programming language that has long dominated enterprise server-side development, is facing a profound restructuring of its technological ecosystem [2]. On one hand, traditional frameworks represented by Spring Boot remain the preferred choice for most enterprise core systems due to their mature ecosystems and broad community support [3]. On the other hand, emerging cloud-native frameworks such as Quarkus and Micronaut have achieved breakthroughs in startup speed and resource efficiency through technologies like AOT(Ahead-of-Time) compilation and memory optimization, offering new possibilities for high-dynamic scenarios such as serverless architectures and edge computing [4]. However, the rapid diversification of technology stacks has also brought significant challenges to enterprise architectural selection: how to balance performance, cost, maintainability, and technological evolution trends has become a pressing practical issue [5].
Academia and industry have extensively researched the technical characteristics of Java frameworks. Existing achievements primarily fall into two categories: first, performance testing and functional analysis of individual frameworks, such as in-depth studies on the Spring ecosystem [6], explorations of Quarkus' native compilation mechanisms [7], and analyses of Micronaut's lightweight architecture [8]; second, a focus on the evolution trends of microservices architecture, including service governance models [9], distributed transaction processing [10], containerized deployment strategies [11], among others. Additionally, in the context of cloud-native transformation, research has addressed the modernization paths of traditional applications and the adaptability evaluation of emerging frameworks in enterprise practices[12]. However, most existing studies emphasize horizontal comparisons or vertical descriptions of technical aspects, lacking a macro-level perspective on evolutionary paths to construct a selection methodology. In particular, there is a lack of decision-support models that integrate technological evolution logic, multi -dimensional evaluation systems, and actual enterprise transformation costs.
The marginal contributions of this paper are mainly reflected in the following three aspects: First, it proposes a three-dimensional "scenario- driver- paradigm" analysis model, placing technological evolution within the macro context of enterprise digital transformation to reveal the business drivers and technical constraints behind architectural changes. Second, it constructs a quantitative selection framework based on dual-track "steady-state- agile-state" adaptation, integrating multiple dimensions such as development efficiency, runtime performance, and cloud-native compatibility to provide enterprises with an operable evaluation tool. Finally, it introduces technical debt and transformation costs as core decision variables, establishing a path-planning model from current state assessment to target architecture to assist enterprises in achieving progressive, low-risk architectural evolution. This paper aims to provide theoretical foundations and practical references for enterprises to make scientific and forward-looking framework selections in complex technological ecosystems through systematic path analysis and structured decision support.

2 MAINSTREAM JAVA ENTERPRISE DEVELOPMENT FRAMEWORKS

2.1 Spring Boot: The Cornerstone of Modern Java Development

In the current landscape of Java enterprise development, Spring Boot undoubtedly occupies a dominant position. Built upon the robust Spring Framework and guided by the core principle of "convention over configuration", it fundamentally addresses the issues of cumbersome configuration and chaotic dependency management prevalent in traditional Spring applications.
The key features of Spring Boot include: First, auto-configuration. The framework automatically configures Spring Beans based on the JAR files present in the classpath—for instance, configuring a data source upon detecting a MySQL driver. Second, starter dependencies. By offering a series of Starter POMs, it groups and packages commonly used dependencies, thereby simplifying build configurations for Maven or Gradle. Third, embedded containers. It integrates directly with servers like Tomcat, Jetty, or Undertow, enabling web applications to be packaged as standalone executable JAR files, which greatly facilitates the deployment and distribution of microservices.The schematic diagram of Spring Boot's auto-configuration mechanism is shown in Figure 1.

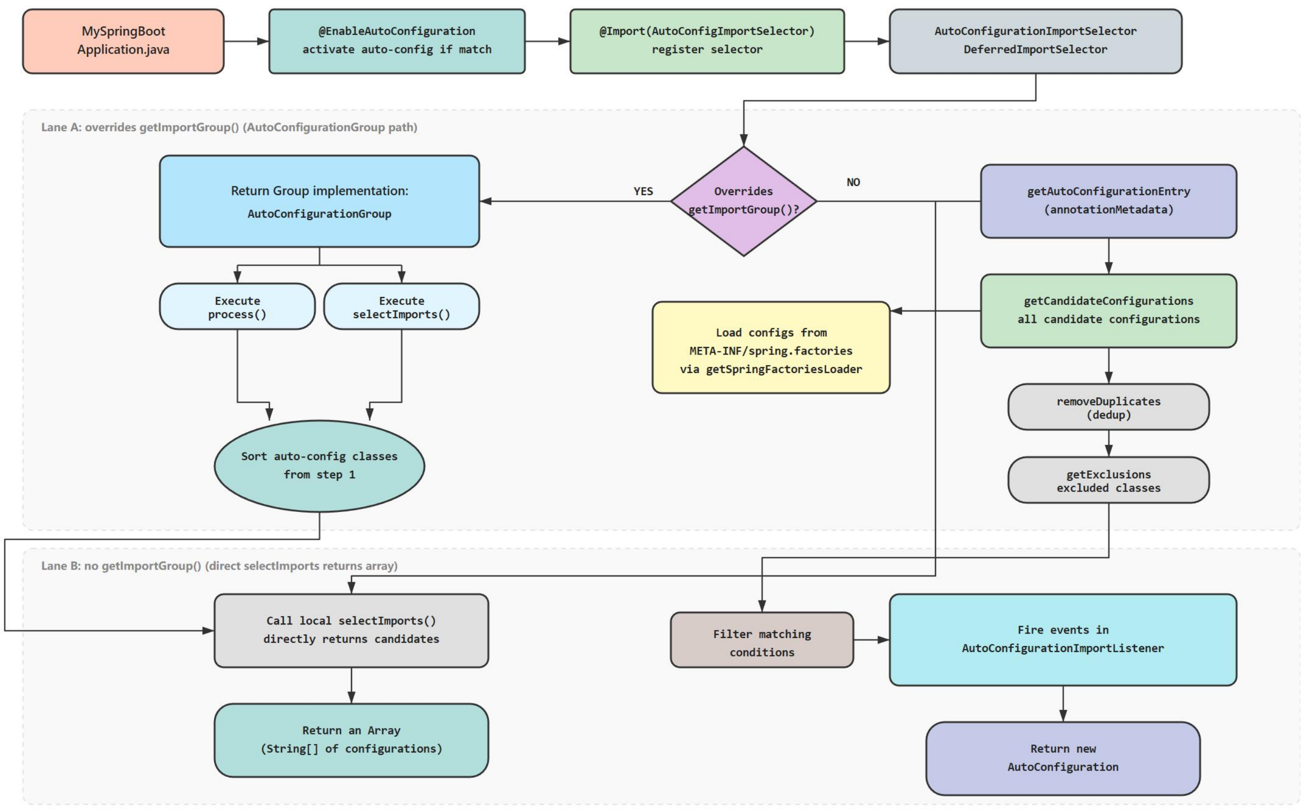
MySpringBoot Application.java
@EnableAutoConfiguration activate auto-config if match
@Import(AutoConfigImportSelector) register selector
AutoConfigurationImportSelector DeferredImportSelector
Lane A: overrides getImportGroup() (AutoConfigurationGroup path)
Return Group implementation: AutoConfigurationGroup
YES
Overrides getImportGroup()?
NO
getAutoConfigurationEntry (annotationMetadata)
Execute process()
Execute selectImports()
Load configs from META-INF/spring.factories via getSpringFactoriesLoader
getCandidateConfigurations all candidate configurations
removeDuplicates (dedup)
Sort auto-config classes from step 1
getExclusions excluded classes
Lane B: no getImportGroup() (direct selectImports returns array)
Call local selectImports() directly returns candidates
Filter matching conditions
Fire events in AutoConfigurationImportListener
Return an Array (String[] of configurations)
Return new AutoConfiguration

Figure 1 Schematic Diagram of Spring Boot Auto-Configuration Mechanism

Its primary advantage lies in significantly lowering the entry barrier and reducing development time for Spring-based applications, allowing developers to focus on implementing business logic. Currently, the overwhelming majority of newly initiated Java projects adopt Spring Boot as their foundational framework. Its application prospects remain extensive, and it stands as an indispensable foundational component for constructing microservices architectures.

2.2 Spring Cloud: An Integrated Solution for Microservices Governance

As monolithic applications evolve toward distributed microservices architectures, challenges such as inter-service communication, fault tolerance, and configuration management have become increasingly prominent. Spring Cloud provides a comprehensive suite of microservices solutions built on Spring Boot, integrating mature industry components such as Netflix OSS and Alibaba.
Research indicates that the core technical components of Spring Cloud include: service discovery and registration centers (e.g., Eureka, Nacos) for managing the status of service instances; load balancers (e.g., Ribbon, Spring Cloud LoadBalancer) for client-side traffic distribution; circuit breakers (e.g., Hystrix, Sentinel) to prevent service avalanche effects; and API gateways (e.g., Zuul, Gateway) to unify external request entry points.
Spring Cloud is characterized by its mature ecosystem, rich set of components, and deep integration with the Spring framework. For medium to large-scale internet enterprises, it offers distributed system governance capabilities that have

been rigorously validated in large-scale production environments. Although Service Mesh technology is gaining traction, Spring Cloud is expected to remain the mainstream choice for microservices architecture in the Java domain for the foreseeable future.

2.3 Persistence Layer Frameworks: The MyBatis vs. Hibernate Dilemma

In the domain of data persistence, the evolution of business models in the internet era has driven an architectural shift from monolithic databases to distributed relational databases. This transition has witnessed the emergence of beneficial practices such as database sharding and NewSQL. This transformation in the underlying storage architecture imposes higher demands on middleware, resulting in two mainstream frameworks—MyBatis and Hibernate—representing two distinct design philosophies.

Hibernate, as a comprehensive fully automated ORM (Object-Relational Mapping) framework, aims to construct a complete persistence container. It utilizes HQL (Hibernate Query Language) to abstract underlying database differences, thereby enabling an object-oriented development paradigm. However, in distributed systems involving complex join queries or performance-sensitive scenarios, its "black-box" mechanism often leads to difficulties in SQL tuning.

In contrast, MyBatis adopts a "semi-automated" lightweight design paradigm, placing SQL statements at the core of its configuration and allowing developers to directly author and optimize native SQL. This "SQL-first" strategy not only facilitates more efficient collaboration with DBAs for index optimization but also offers greater flexibility in adapting to the routing rules of distributed middleware. Consequently, depending on the business emphasis—whether prioritizing "development convenience" or "runtime performance"—these two frameworks occupy significantly different market segments: MyBatis dominates high-concurrency internet systems in China, whereas Hibernate remains more prevalent in European and American markets as well as traditional enterprise information systems.

As a fully automated ORM framework, Hibernate can map Java objects to database tables through straightforward configuration and automatically generate SQL statements. Its advantages include high development efficiency and excellent database portability. Nevertheless, in scenarios involving complex join queries or requiring extreme performance optimization, the SQL automatically generated by Hibernate is often difficult to control and is prone to the N+1 query problem.

MyBatis employs a semi-automated object-relational mapping design, decoupling SQL logic from program code and granting developers fine-grained control over database interaction processes and execution plans. Its built-in dynamic SQL construction mechanism effectively aligns with the stringent constraints on complex query latency and throughput in high-concurrency scenarios. It supports deep index optimization and execution path tuning for specific business logic, thereby mitigating the uncontrollable performance risks associated with fully automated frameworks when generating complex join queries.

The MyBatis-Plus extension component further addresses ecosystem limitations. While preserving the flexibility of native SQL, it provides automated generation capabilities for standardized common data operations. This technology stack model, which harmonizes underlying performance control with high-level development efficiency, achieves a balance between engineering quality and delivery speed, establishing its de facto standard status in the development of core enterprise business systems in China.

3 EMERGING FRAMEWORK TECHNOLOGIES AND CLOUD-NATIVE TRENDS

3.1 Challenges of the Cloud-Native Era

Kubernetes has established itself as the de facto standard in container orchestration, propelling cloud-native infrastructure toward fine-grained, highly dynamic evolution. Traditional Java Virtual Machines (JVMs), reliant on dynamic class loading and just-in-time (JIT) compilation mechanisms, prioritize peak throughput for long-running processes by design. This inevitably entails significant warm-up overhead during startup and substantial resident memory consumption.

Serverless architectures and edge computing scenarios impose stringent demands on the instantaneous elasticity and deployment density of computing resources. Short-lived function execution instances require millisecond-level startup speeds, while resource-constrained edge nodes enforce strict memory quotas for individual services.

The sluggish cold-start performance of Java applications in such heterogeneous environments directly contributes to tail latency degradation in service responses. The high baseline memory resource footprint significantly escalates cloud computing costs under usage-based billing models and increases operational overhead.

3.2 Quarkus: Built for Kubernetes

STo address these pain points, Red Hat introduced the Quarkus framework. Its tagline, "Supersonic Subatomic Java," underscores its core technology: leveraging GraalVM(Graal Virtual Machine) to compile Java applications into native binary files, known as Native Images.Performance comparison: Quarkus versus traditional frameworks, as shown in Figure 2.

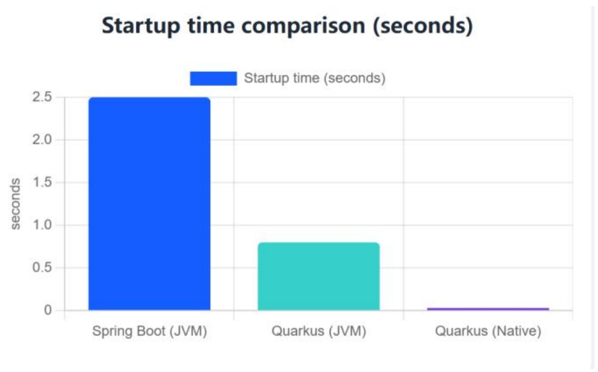

Figure 2 Performance Comparison: Quarkus vs. Traditional Frameworks

The Quarkus framework adopts a compile-time boot design paradigm. By utilizing GraalVM native image technology, it shifts intensive framework initialization, classpath scanning, and dependency injection processes to the build phase. This mechanism achieves sub-second cold-start latency and delivers an order-of-magnitude optimization in memory footprint, with runtime memory usage being merely one-tenth of that in traditional OpenJDK environments. This effectively overcomes the physical limitations of Java in resource-constrained container environments.

Regarding developer experience, Quarkus implements an instant coding mechanism based on classloader isolation, enabling hot deployment for business logic modifications and significantly shortening the development-testing feedback loop. In terms of ecosystem compatibility, the framework strictly adheres to Jakarta EE standards such as JAX-RS(Java API for RESTful Web Services) and CDI(Contexts and Dependency Injection), ensuring low-cost migration of existing technical assets to cloud-native architectures.

Quarkus demonstrates notable performance advantages and resource utilization potential in compute-intensive tasks and high-frequency dynamic scaling scenarios within serverless architectures. It provides a critical evolutionary pathway for the Java technology stack in the post-cloud-native era.

3.3 Micronaut: Compile-Time Dependency Injection

The Micronaut framework delivers a full-stack microservices solution based on ahead-of-time compilation technology. Distinct from the traditional Spring ecosystem, which relies on runtime reflection mechanisms for dynamic proxying, this framework employs Java annotation processors to pre-construct dependency injection topologies and aspect-oriented proxy logic during the compilation phase. This design paradigm entirely eliminates the runtime computational overhead associated with reflection operations. It decouples the relationships between application startup time, heap memory consumption, and codebase size, achieving a physical separation where performance metrics become independent of business logic complexity.

Micronaut natively integrates microservices governance components, including service discovery, distributed tracing, and circuit breakers. This architecture enables the construction of highly available distributed systems without depending on heavyweight third-party middleware, significantly reducing deployment complexity and maintenance costs in heterogeneous cloud environments. For enterprise core systems seeking to shed the historical technical debt of traditional frameworks and pursue extreme startup speeds and low-latency response characteristics, Micronaut offers an alternative evolutionary path that combines lightweight deployment with high-performance computing capabilities.

4 EVOLUTION LOGIC AND SELECTION STRATEGY FOR JAVA FRAMEWORK TECHNOLOGIES

4.1 Paradigm Shift from "Runtime" to "Compile-Time"

Throughout the developmental history of Java frameworks, a clear evolutionary trajectory is evident: from the early era of complex configuration (the XML age), to Spring Boot's principle of "convention over configuration," and now to the "compile-time priority" approach of frameworks like Quarkus. The traditional Spring ecosystem heavily depends on the JVM's dynamic proxy and reflection mechanisms. While this affords considerable development flexibility, the issues of slow startup speeds and high memory consumption have become increasingly pronounced in cloud-native environments. The new generation of frameworks (e.g., Quarkus) employs AOT (Ahead-of-Time) compilation technology, shifting operations such as dependency injection and class loading—traditionally performed at runtime—to the compilation phase. This architectural paradigm shift, essentially a "space-for-time" trade-off, is fundamentally driven by the need to meet the stringent demands for rapid scaling and high-density deployment in containerized environments like Kubernetes.

4.2 Technical Selection Recommendations for Hybrid Architectures

Given the current technological landscape, enterprises should avoid blindly pursuing new technologies in their architectural selections. Instead, a differentiated hybrid strategy is advisable:

Core Business and Complex Transactional Systems: It is recommended to continue utilizing the Spring Boot/Spring Cloud ecosystem. Its mature lifecycle, comprehensive middleware support, and vast developer community can maximize system stability and maintainability.
Serverless, Edge Computing, and High-Density Microservices: Experimentation with Quarkus or Micronaut is recommended. In resource-constrained scenarios or those requiring extreme startup speeds, these natively compiled frameworks can significantly reduce cloud resource costs (the "Cloud Bill").
Persistence Layer Selection: For internet applications with complex business logic and high demands for SQL tuning, MyBatis remains the preferred choice. For internal management systems with standardized business models that prioritize rapid iteration, Hibernate (JPA) offers greater efficiency advantages.

5 CONCLUSIONS

This paper systematically analyzes the technological evolution path of Java enterprise-level Web development frameworks, constructing a selection methodology based on the three-dimensional "scenario-driver-paradigm" analysis model and a dual-track "steady-state-agile-state" adaptation framework. The study reveals a clear divergence in the current technological ecosystem: Spring Boot/Cloud continues to dominate traditional enterprise application development with its mature ecosystem, while cloud-native frameworks such as Quarkus and Micronaut have achieved breakthroughs in startup performance and resource efficiency, offering superior solutions for emerging scenarios like Serverless and edge computing. The hybrid architecture selection strategy proposed in this study comprehensively considers multidimensional factors including technical performance, transformation costs, and organizational capabilities, providing enterprises with a progressive migration path from current state assessment to target architecture, demonstrating strong feasibility in practical applications.
Future Java framework technology will continue to evolve toward intelligence, heterogeneity, and security-native design. On one hand, technological innovations such as AI compilation optimization and low-code integration will further enhance development efficiency. On the other hand, inherent support for green computing and zero-trust architecture will become critical considerations in framework design. Enterprises need to actively explore the application boundaries of cloud-native technologies while maintaining system stability. By adopting a dual-mode strategy of "steady-state operation for core systems and agile-state piloting for innovative services," they can leverage digital competitive advantages amid technological iterations. Subsequent research may further conduct large-scale empirical analyses to dynamically track the evolutionary trends of framework technologies, providing more precise selection guidance for the industry.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Yin Zhen, Lee SUJ. Security Analysis of Web Open-Source Projects Based on Java and PHP. Electronics, 2023, 12(12): 2618.
[2] Hou Xiangfeng, Wang Yihao. Design and Application of Basketball Microservice Platform. Mobile Information Systems, 2022: 3928363.
[3] Lv Taizhi, Tang Peiyi, CHEN Yingying. Research and Practice on Project based Teaching of Java Web Application Development Course based on Shipping Characteristics. Frontiers in Humanities and Social Sciences, 2022, 2(10).
[4] Di FP, Paolone G, Paesani R, et al. Design and Implementation of a Metadata Repository about UML Class Diagrams. A Software Tool Supporting the Automatic Feeding of the Repository. Electronics, 2022, 11(2): 201.
[5] Moia GS, Gaia ASC, Oliveira MSD, et al. ReNoteWeb – Web platform for the improvement of assembly result and annotation of prokaryotic genomes. Gene, 2022, 844: 146819.
[6] Agun HV. WebCollectives: A light regular expression based web content extractor in Java. SoftwareX, 2023, 24: 101569.
[7] Xu Dishi, Liu Fagui, Wang Bin, et al. GenesisRM: A state-driven approach to resource management for distributed JVM web applications. Future Generation Computer Systems, 2025, 163: 107539.
[8] Pleciński P, Bokla N, Klymkovych T, et al. Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks. Sensors, 2022, 22(20): 7759.
[9] Wei Yinhong, Niu Weina, Shen Yi, et al. WIVIM: Web Injection Vulnerabilities Detection Based on Interprocedural Analysis and MiniLM-GNN. Peer-to-Peer Networking and Applications, 2025, 18(1): 245.
[10] Xu Ke, Zhang Bing, Li Jingyue, et al. DRecv: Detecting and auto-repairing vulnerabilities in role-based access control in web application. Journal of Network and Computer Applications, 2025, 240: 104191.
[11] Wang Shiyu, Jiang Yuyao, Xu Shuaijianni, et al. Fine-tuning a vulnerability-specific large language model for a hybrid software vulnerability detection method. Engineering Applications of Artificial Intelligence, 2026, 165: 113410.
[12] Setyautami MRA, Adianto D, Azurat A, et al. A PROPOSED JAVA WEB FRAMEWORK TO SUPPORT SOFTWARE PRODUCT LINE ENGINEERING. ICIC Express Letters, 2024, 18(3): 293-301.