



Methodology for Developing and Optimizing Web Applications Based on Java and JavaScript Technologies

Olena Komleva
Independent Researcher
Chicago, Illinois
United States of America
ORCID: 0009-0002-2796-6449

ABSTRACT

Purpose. This article develops an integrated methodology for building and optimizing web applications with Java and JavaScript, treating architecture, testing, runtime analysis, and iterative refinement as parts of a single engineering cycle rather than as isolated tasks. **Methodology.** The study employs a qualitative meta-synthesis design that aggregates empirical evidence from heterogeneous sources. Twenty verified English-language publications were synthesized, including systematic reviews, empirical software engineering papers, industrial case studies, and foundational works on Java, JavaScript, testing, and performance. A compact secondary analytical layer compares recurring optimization factors, testing practices, and quality indicators across the selected studies. **Findings.** Sustainable performance gains emerge when server-side decisions, client-side behavior, observability, and automated testing are aligned early in the lifecycle. Bottlenecks in web systems rarely originate in one layer. They are usually produced by interactions among backend configuration, frontend execution paths, dependency behavior, and incomplete testing coverage. A profile of the corpus by period and venue and an application to three project scenarios extend the synthesis. **Unique contribution to theory, policy, and practice.** The article proposes an authorial practical framework, the Integrated Dual-Layer Optimization Cycle, which translates empirical findings into a replicable workflow for Java and JavaScript web projects. The framework addresses researchers, educators, and practitioners who need a methodologically coherent way to connect development choices with maintainability, scalability, and measurable optimization outcomes.

General Terms

Software Engineering, Performance Optimization, Testing and Verification, Measurement, Web Systems, Design Methodology.

Keywords

Java; JavaScript; web application development; web application optimization; software engineering methodology; automated testing; performance engineering; observability; maintainability; empirical synthesis.

1. INTRODUCTION

Modern web applications are built in a technological environment where the decoupling of development and optimization phases has become computationally and economically inefficient. A system may satisfy functional

requirements at launch and still fail under realistic traffic patterns, unstable client-side execution, or an accumulation of inefficient implementation choices that were never critical in isolation. This tension is especially visible in projects that combine Java on the server side and JavaScript in the browser or in an adjacent client runtime. Java remains one of the most reliable foundations for scalable business logic, transactional consistency, and service integration. JavaScript continues to dominate interactive web execution, interface orchestration, and increasingly complex application behavior in the client layer.

The widespread use of both technologies has not removed methodological fragmentation. It has often reinforced it. Teams still discuss backend performance, frontend responsiveness, and testing maturity as though they belong to different conversations, even when end-user quality depends on their interaction. Foundational studies on JavaScript behavior and analysis showed that the dynamic nature of the language complicates reasoning about correctness and performance in ways that traditional assumptions cannot fully capture [1], [2]. Research on automated testing demonstrated early that modern web systems cannot be validated adequately through manual test case construction alone, because web execution paths are stateful, event-driven, and combinatorially large [3]. A systematic review of web application testing later confirmed that the field had expanded rapidly but remained methodologically heterogeneous, especially in the alignment of tools, layers, and evaluation criteria [4]. Empirical work on JavaScript performance made it clear that many costly slowdowns are not exotic defects but ordinary engineering decisions that become visible only under realistic workloads [5].

Web optimization is therefore not a final polishing stage. It is an architectural and methodological problem that begins before deployment and persists throughout the software lifecycle. This article addresses that problem by arguing that web applications based on Java and JavaScript should be designed and improved through a unified methodology that synchronizes architectural planning, testing, observability, performance inspection, and controlled iteration.

The rationale for this study lies in a practical gap. Existing research offers rich knowledge about single layers, single metrics, or single practices, yet practitioners frequently need a disciplined way to combine them. The objective of the article is not to claim a new universal framework detached from prior evidence, but to derive a coherent methodological model from verified empirical and analytical studies. No primary empirical



experiment was conducted. The article instead performs a structured synthesis of twenty sources and uses that synthesis to formulate a small secondary empirical layer, namely a comparative extraction of recurring patterns across earlier empirical investigations. The core research question is stated simply: which methodological elements most consistently support the development and optimization of Java and JavaScript web applications, and how can these elements be integrated into a practical authorial methodology suitable for real projects?

2. LITERATURE REVIEW

The selected literature reveals a stable but multi-layered research landscape. One cluster concerns the internal behavior of JavaScript and the challenge of analyzing it with sufficient precision. Richards et al. [1] showed through large-scale dynamic observation that JavaScript in the wild often violates simplifying assumptions commonly made in tools and research models. Sun and Ryu [2] later synthesized the broader analytical problem space and demonstrated that dynamism, prototype behavior, reflective features, and framework-heavy ecosystems continue to complicate static and hybrid analysis. This line of work matters methodologically because optimization cannot begin with measurement alone. It also depends on how reliably engineers can reason about actual program behavior. Related research proposed stronger analytical mechanisms for JavaScript-rich applications. Jensen et al. [6] introduced type analysis for JavaScript, while Madsen et al. [7] extended practical static analysis into framework-intensive and library-intensive settings, showing that realistic analysis becomes more useful when it models ecosystem dependencies instead of treating them as external noise.

A second cluster centers on testing, which is arguably the most visible bridge between development and optimization. Artzi et al. [3] presented a framework for automated testing of JavaScript web applications that directly addressed the inadequacy of purely manual test construction in dynamic web environments. Doğan et al. [4] provided a systematic review confirming that web testing had become broad, but not necessarily integrated, with studies often focusing on specific defects, tools, or levels of abstraction. The field has not become simpler with time. Balsam and Mishra [8] argue that contemporary web application testing still faces persistent challenges around heterogeneity, automation scope, and quality trade-offs. Evidence from open-source repositories adds another layer. Di Meglio et al. [9] demonstrate that automated web GUI testing has become more widespread and operationally significant, but its adoption and maintenance depend on framework choice, co-evolution practices, and the discipline with which teams treat tests as first-class assets. If testing is treated as an afterthought, optimization becomes reactive. If it is embedded in the lifecycle, optimization becomes measurable.

A third cluster concerns performance and runtime efficiency. Selakovic and Pradel [5] showed that JavaScript performance problems often arise from ordinary coding decisions and that substantial optimization is frequently possible through relatively contained changes. On the Java side, Noetzold et al. [10] demonstrated empirically that JVM configuration remains a consequential factor in web application performance, especially when tuning decisions are aligned with real workload characteristics rather than with default assumptions. Industrial evidence reinforces the same lesson from another angle. Van Riet et al. [11] describe optimization as a continuous activity embedded in delivery rather than a one-time

intervention, while Shatnawi et al. [12] argue through an industrial case study that observability is no longer optional in web systems, because distributed execution and production uncertainty make invisible degradation too expensive. Performance is therefore not reducible to speed. It emerges from architecture, runtime instrumentation, deployment conditions, and disciplined interpretation of signals.

Several studies address narrower but methodologically valuable aspects of quality evolution. Wang et al. [13] analyze bugs in JavaScript engines and show that execution platforms themselves shape the defect surface experienced by developers. Jiang et al. [14] examine co-changed entities in JavaScript programs and demonstrate that change patterns are often structurally related, which means maintainability and optimization should be understood through dependency-aware evolution rather than through isolated edits. Rack and Staicu [15] focus on JavaScript bundling and its security implications, a reminder that optimization techniques which reduce delivery overhead may also introduce new risks or opacity. Yan et al. [16] contribute a comparative lens through their work on WebAssembly applications. They show that performance discussions on the web cannot rely on slogans about one technology being faster than another, because actual results depend on workload shape, browser behavior, and execution context. This discourages one-dimensional optimization narratives in projects that combine Java and JavaScript.

Java-focused empirical studies extend the same logic. Leitner and Bezemer [17] found that performance testing practice in Java-based open-source projects was far from universal, which suggests that many systems still enter production without a mature culture of non-functional validation. Imran et al. [18] investigated code coverage in performance testing and showed that the relationship between coverage and meaningful non-functional assurance is more complex than teams often assume. Liu et al. [19] evaluated automated test generation tools for Java and identified both effectiveness gains and practical limitations, a reminder that automation is an enabler rather than a substitute for methodological judgment. Spadini et al. [20] examined the use of mock objects in Java systems and highlighted the trade-off between isolation and coupling in test design.

Taken together, the literature indicates a clear gap. There is no shortage of research about individual components of development and optimization. What remains comparatively underdeveloped is a methodologically compact framework that treats Java backend engineering, JavaScript execution, testing maturity, and observability as a single lifecycle problem. This article responds to that gap by moving from literature accumulation to methodological integration.

3. METHODOLOGY

This study uses a qualitative-analytical research design grounded in secondary empirical synthesis. No primary experiment, field deployment, or controlled intervention was conducted for this publication. Twenty English-language sources were selected because their titles, bibliographic identities, and scholarly existence could be verified through reliable publisher or indexing records. The corpus intentionally combines foundational studies, recent peer-reviewed journal articles, conference papers, and industrial case studies, so that the resulting methodology would not be biased toward a single research culture.

The analytical procedure consisted of four consecutive stages. First, the studies were grouped by dominant contribution area:

program analysis, testing, performance optimization, maintainability and evolution, and runtime observability. Second, each source was coded according to the main methodological variable it foregrounded, such as execution behavior, test automation, configuration, bug patterns, GUI testing adoption, code coverage, or infrastructure-dependent performance. Third, cross-study comparison identified recurring relationships among variables rather than merely recurring keywords. Fourth, the extracted relationships were translated into a practical authorial methodology intended for

projects that combine Java and JavaScript.

Table 1 reports the analytical profile of the corpus along four coding dimensions and shows how the twenty studies distribute across contribution area, technological layer, validation mode, and lifecycle position. Shares exceed one hundred per cent within the technological layer dimension because cross-layer studies were counted in both the backend and the frontend perspective where their evidence applied to each side.

Table 1. Analytical profile of the 20-source corpus

Dimension	Category	Count	Share
Contribution area	Program analysis	4	20%
Contribution area	Testing and quality assurance	6	30%
Contribution area	Performance optimization	4	20%
Contribution area	Maintainability and evolution	3	15%
Contribution area	Runtime observability	3	15%
Technological layer	Backend-focused	4	20%
Technological layer	Frontend-focused	7	35%
Technological layer	Cross-layer	9	45%
Validation mode	Empirical studies	11	55%
Validation mode	Systematic reviews and surveys	3	15%
Validation mode	Industrial case studies	3	15%
Validation mode	Analytical and foundational studies	3	15%
Lifecycle position	Design-time	5	25%
Lifecycle position	Test-time	6	30%
Lifecycle position	Deployment-time	3	15%
Lifecycle position	Runtime	6	30%

Two schematics summarize the shape of the corpus. The first is a bar chart of contribution areas, in which the horizontal axis carries the five coded areas and the vertical axis carries the number of studies assigned to each. It shows that testing and quality assurance form the largest single block, while program analysis and performance optimization occupy comparable second positions, and maintainability and observability form smaller but non-marginal groups (Fig. 1). The distribution matters for the argument because it establishes that the proposed methodology rests on a corpus in which no single concern dominates to the point of distortion.

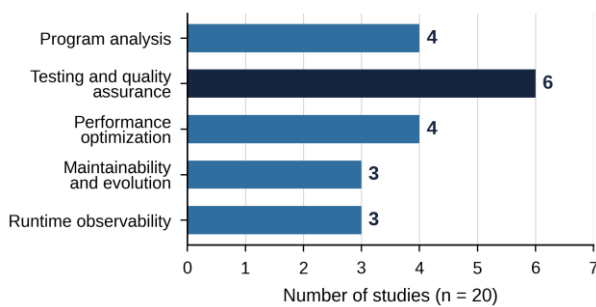


Fig 1: Distribution of reviewed studies by contribution area

The second schematic maps the same corpus onto the software lifecycle. Its four categories, design-time, test-time, deployment-time, and runtime, are plotted against the number of studies that address each stage, which makes the coverage asymmetry directly visible: test-time and runtime evidence together account for the majority of the corpus, whereas deployment-time evidence is comparatively thin (Fig. 2). This asymmetry justifies a cycle that treats verification and instrumentation as load-bearing phases rather than as optional supplements.

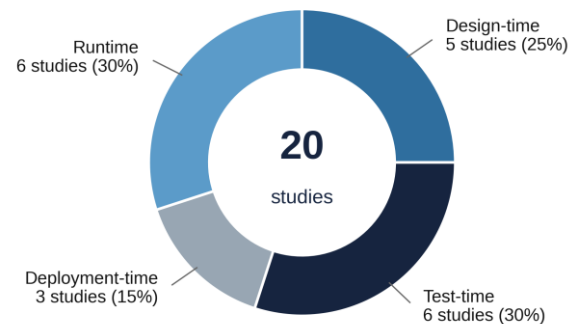


Fig 2: Lifecycle coverage of the reviewed corpus

Although the article does not include a traditional empirical dataset, it contains a limited comparative empirical component in the form of analytical aggregation. The paper treats prior empirical studies as evidence units. This allows the research to answer a modest but meaningful question: which methodological features recur most consistently when successful development and optimization are discussed in verified empirical literature?

Five comparison criteria were applied across the corpus. The first was the technological layer, distinguishing backend-oriented, frontend-oriented, and cross-layer studies. The second was quality focus, distinguishing correctness, performance, maintainability, observability, and security. The third was validation mode, separating analytical proposals from empirical investigations and industrial case studies. The fourth was lifecycle position, identifying whether the study addressed design-time, test-time, deployment-time, or runtime behavior. The fifth was actionability, that is, whether the findings could realistically be translated into engineering decisions for web teams.



Three robustness checks complement these criteria: a profile of the corpus by publication period and venue type, a trace of the sources behind each central finding, and an application of the cycle to three project scenarios. All three rely only on the twenty sources and their bibliographic records.

This design was chosen because an article devoted to methodology should not merely summarize prior work. It should transform that work into an ordered approach. The study also acknowledges its limitations. Secondary synthesis cannot claim causal novelty in the way that a controlled experiment can. It is shaped by the selected corpus and by the interpretive lens through which the studies are compared. In a field where practitioners often inherit fragmented guidance from separate communities, a rigorous synthesis-based methodology nevertheless remains scientifically useful. The methodological assumption guiding this article is explicit: if a feature appears repeatedly across independent empirical studies and across both Java and JavaScript contexts, it deserves to be treated not as an incidental tactic but as a structural component of web engineering practice.

4. RESULTS

The comparative synthesis produced three central findings. First, effective optimization begins with integration, not with emergency repair. Studies focused on JavaScript behavior, testing, and runtime analysis repeatedly indicate that problems become costly when they are discovered late and interpreted locally rather than systemically [1], [3], [5]. Second, backend and frontend quality are not independent performance zones. JVM tuning, server architecture, and service orchestration influence the same user-visible outcome space that is also shaped by client-side rendering, bundling, event handling, and browser execution variability [10], [11], [16]. Third, the most practically valuable studies connect software quality with repeatable engineering routines. Repository mining, automated GUI testing, empirical evaluation of tools, and observability-

driven refinement all point toward disciplined iteration rather than ad hoc intervention [9], [12], [17], [19].

Table 2 condenses the synthesized findings, the number of sources that support each of them, and the practical implication that follows for engineering teams.

When the corpus was recoded by methodological signal, testing-related evidence appeared in the majority of sources (15 of 20 support testing as an optimization enabler), and performance was the most frequent quality dimension (17 of 20 sources, Fig. 4).

Three further analyses test how far these findings depend on the corpus and on project context: Table 3 profiles the corpus by period and venue, Table 4 traces the sources behind each central finding, and Table 5 applies the cycle to three scenarios.

The frequency chart plots each synthesized finding against the number of reviewed studies that support it, which allows the relative evidential weight of integration, cross-layer dependence, testing, observability, and evolutionary control to be compared at a glance (Fig. 3). The near parity of the top bars is the analytical point: many teams still separate testing from optimization, and the reviewed studies do not support that separation. Testing is the mechanism that makes optimization trustworthy. Observability is the mechanism that makes it sustainable after release.

A complementary schematic decomposes the corpus by quality dimension, showing how correctness, performance, maintainability, observability, and security are represented in the evidence base (Fig. 4). It clarifies that no single quality attribute can act as a proxy for the others, which is the structural reason why the proposed cycle assigns a dedicated phase to verification, to instrumentation, and to evolutionary control instead of collapsing them into a single quality gate.

Table 2. Synthesized findings and practical implications

Synthesized finding	Supporting studies	Practical implication
Integration-first optimization	16	Optimization should begin with architectural coordination rather than post-release repair.
Cross-layer performance dependence	14	The Java backend and the JavaScript frontend should be optimized as one interacting system.
Testing as an optimization enabler	15	Automated tests make optimization decisions measurable and trustworthy.
Observability for sustainable improvement	12	Runtime telemetry should guide iterative refinement after release.
Evolutionary control of changes	10	Optimization decisions must be checked for maintainability and security side effects.
Integrated Dual-Layer Optimization Cycle	6	The authorial method condenses the synthesis into a six-phase repeatable workflow.

Table 3. Corpus composition by publication period and venue type

Publication period	Sources	Journal articles	Conference papers	Total	Share
2009 to 2017	[1] to [7], [17]	2	6	8	40%
2019 to 2026	[8] to [16], [18] to [20]	9	3	12	60%
All sources	[1] to [20]	11	9	20	100%

Table 4. Period profile of the sources cited for each central finding

Central finding	Sources cited in the Results	2009 to 2017	2019 to 2026
Integration-first optimization	[1], [3], [5]	3	0
Cross-layer performance dependence	[10], [11], [16]	0	3
Repeatable engineering routines	[9], [12], [17], [19]	1	3

Table 5. Application of the cycle to three project scenarios (analytical profiles)

Project scenario	Dominant risk	Priority phases of the cycle	Evidence
Legacy Java monolith with a growing JavaScript layer	Weakly tested, dynamically behaving client code	2 (verification-first development), then 3 (runtime transparency)	[1], [3], [7], [17]
New Java services with a framework-heavy single-page client	Cross-layer latency, default JVM settings, bundle growth	1 (architectural alignment), then 4 (targeted optimization)	[5], [10], [15], [16]
Production system under load with limited telemetry	Degradation that stays invisible until users report it	3 (runtime transparency), then 5 (evolutionary control)	[11], [12], [14]

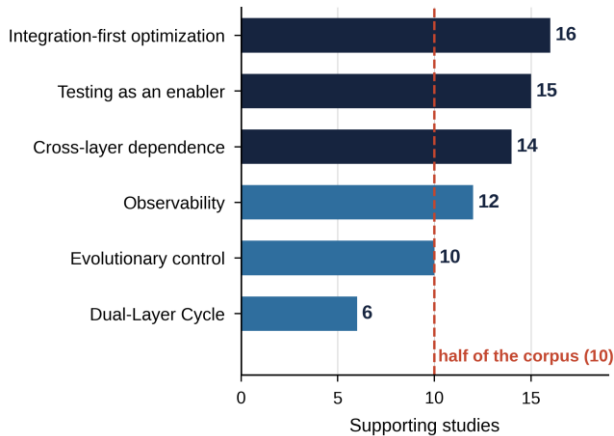


Fig 3: Frequency of key synthesized findings across the reviewed studies

A secondary finding emerged from narrower empirical works. Repository-level and language-level studies suggest that change behavior, bug distribution, and bundling practices are not peripheral concerns. They influence the maintainability envelope within which optimization remains economically feasible [13], [14], [15]. Java-specific testing studies indicate that automation quality depends on what exactly is being automated. Coverage alone does not guarantee useful performance assurance, and excessive abstraction in test design may create long-term maintenance cost instead of reducing it [18], [20].

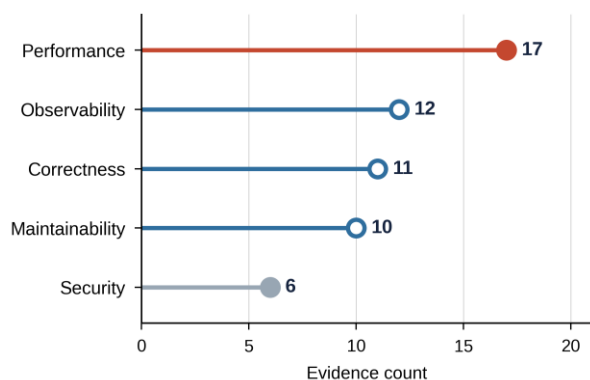


Fig 4: Quality dimensions emerging from the synthesis

Persistent heterogeneity in tooling and evaluation criteria reinforces the same conclusion [8], as does the analytical difficulty of reasoning about framework-heavy client code [6], [7].

Support for the five empirical findings ranges from 80 per cent of the corpus for integration-first optimization (16 of 20 sources) to 50 per cent for evolutionary control (10 of 20), so

all five empirical findings reach the half-corpus threshold shown in Fig. 3. Only the cycle itself, which is a derived method and not a finding, falls below it (6 sources). Cross-layer studies form the largest technological group (9 sources, 45 per cent), ahead of frontend-focused (7) and backend-focused (4) studies, and test-time and runtime evidence together cover 12 sources (60 per cent), against 3 for deployment time. The counts in Fig. 4 sum to 56 for 20 sources (2.8 per source), so the typical study addresses several quality attributes at once. Performance is the most frequent dimension (17 sources) and security the least frequent (6).

Table 3 shows that the corpus does not depend on a single period or publication culture. Twelve of the twenty sources (60 per cent) appeared between 2019 and 2026 and eight (40 per cent) between 2009 and 2017. The older block is dominated by conference papers (6 of 8), and the recent block by journal articles (9 of 12).

Table 4 breaks the three central findings down by the period of the sources cited for them. The evidence for cross-layer dependence is entirely recent (three of three sources from 2019 to 2026), whereas the sources cited for integration-first optimization all belong to the earlier period (three of three from 2009 to 2017). The third finding draws on both periods, with one earlier and three recent sources. Integration-first optimization is therefore the finding most exposed to the objection that it rests on older evidence, although a recent industrial case study treats optimization as embedded in delivery [11].

Table 5 applies the cycle to three project scenarios that differ in their dominant risk. The scenarios are analytical profiles built from the cited evidence, not experimental cases. In a legacy monolith with a growing JavaScript layer, dynamic and weakly tested client code [1], [3], [7] and sparse performance testing [17] make verification-first development the first priority. In a new service-oriented system with a framework-heavy client, ordinary JavaScript performance defects [5], default JVM settings [10], bundle composition [15], and workload-dependent execution costs [16] are the main risks, so architectural alignment precedes targeted optimization. In a production system with limited telemetry, degradation stays invisible without instrumentation [11], [12], and dependency-aware change analysis [14] indicates which edits must travel together. The phase order differs between scenarios, while the loop stays fixed.

4.1 The Integrated Dual-Layer Optimization Cycle

On the basis of these findings, the article proposes a concise authorial methodology: the Integrated Dual-Layer Optimization Cycle. The framework functions as a methodological abstraction layer that formalizes the transition between discrete development stages. Its first phase is architectural alignment. The Java backend and the JavaScript

frontend are planned together around request paths, state transitions, data contracts, and likely performance-sensitive interactions. Its second phase is verification-first development. Functional tests, integration tests, and web GUI tests are introduced as parallel design instruments, not only as post hoc validators. Its third phase is runtime transparency, which requires teams to instrument the application before serious optimization begins, so that later performance decisions rest on evidence rather than on intuition.

The schematic of the cycle presents the six phases as a closed

loop rather than as a linear pipeline. Its components are the two technology layers, drawn as parallel tracks, the phase nodes that sequence architectural alignment, verification-first development, runtime transparency, targeted optimization, evolutionary control, and empirical recalibration, and the feedback arrows that return runtime evidence to the design stage (Fig. 5). The diagram is essential to the argument because the contribution of the framework lies in the direction and the ordering of those arrows, not in the individual phases, each of which is already familiar in isolation.

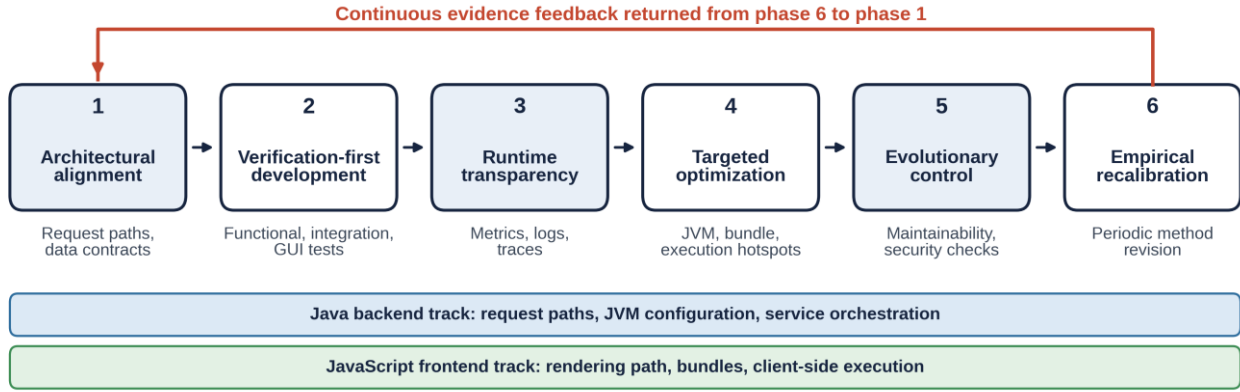


Fig 5: The authorial methodology, the Integrated Dual-Layer Optimization Cycle

4.2 Instrumentation of runtime transparency

The implementation of runtime transparency in the third phase requires a unified observability stack that bridges the Java and JavaScript execution environments. Three instrumentation layers are suggested.

Backend layer. Micrometer and Prometheus provide real-time monitoring of JVM heap health and request latency.

Frontend layer. OpenTelemetry supports distributed tracing, which allows teams to correlate client-side rendering delays with backend service orchestration.

Unified analysis. Shared dashboards such as Grafana visualize the continuous evidence feedback loop, so that every optimization step is guided by production-quality telemetry rather than by developer intuition.

4.3 Parametric tuning in targeted optimization

The fourth phase is targeted optimization. The team prioritizes a small number of hotspots validated through tests and runtime signals, whether they arise from JVM configuration, expensive Java service logic, heavy JavaScript execution, bloated bundles, or unstable client-server interaction. At this point the methodology moves from architectural reasoning to parametric tuning. For the Java backend this requires empirical analysis of JVM configurations aligned with specific workload characteristics. Memory management interventions include transitioning to low-latency garbage collectors such as ZGC or tuning G1GC parameters, for example the maximum pause time target, in order to stabilize response times. On the JavaScript side, optimization focuses on minimizing the critical rendering path through tree shaking and route-based code splitting. By concentrating on such ordinary engineering decisions, teams achieve substantial performance gains with relatively contained code changes.

The fifth phase is evolutionary control, which treats every significant optimization as a maintainability event. The change must be checked against future co-evolution cost, security side effects, and test resilience. A sixth and lighter phase may be added in mature projects: empirical recalibration. Here the team periodically compares recent runtime patterns and test evidence against the original architectural assumptions and revises them.

The methodological novelty of the cycle lies in its sequencing. It does not wait for production failure before linking Java and JavaScript concerns. It does not assume that frontend refinement can compensate for backend inefficiency, or that server tuning alone can rescue poor client execution. The model is authorial, yet deliberately modest. Its purpose is to condense recurring evidence from prior studies into a form that a real development team can adopt without replacing its entire toolchain. The small empirical contribution of this article is therefore not a new benchmark result but a structured comparative conclusion: across verified studies, the most robust route to optimization is not a single technology choice, but a repeatable cross-layer method.

4.4 Formal decision criterion

To formalize the decision-making process within the proposed cycle, the net benefit of a specific optimization task can be modeled through an objective function.

$$G_{total} = \sum_{i \in \{L_{srv}, L_{cli}\}} (\Delta P_i \cdot W_i) - C_{evol} \quad (1)$$

Here the performance gain term denotes the quantified improvement on the server-side layer or the client-side layer. The weighting coefficient reflects the impact of that layer on the overall end-user experience. The evolutionary cost term represents projected maintainability debt and potential security side effects. This representation supports the principle that optimization should be treated as a selective, evidence-based simplification rather than as a generalized activity.



5. DISCUSSION

The synthesis suggests that the central weakness of many web engineering practices is not a shortage of tools but a shortage of methodological coherence. Java and JavaScript are often discussed as if they belonged to different problem domains, even though they shape one application experience and usually one operational budget. The literature does not support such fragmentation. Foundational analysis papers and recent industrial studies alike indicate that quality emerges from interaction. This is why the proposed methodology places as much emphasis on sequencing as on component choice. A well-tuned JVM cannot fully compensate for unstable client execution. A fast frontend cannot conceal chronic server-side latency. A broad test suite is less valuable when it cannot guide prioritization. Observability arrives too late when the architecture was never designed to expose meaningful signals.

The findings also reinforce a conceptual point about optimization. In software practice, optimization is still sometimes framed as a specialized activity reserved for experts or late-stage crises. The reviewed literature points in another direction. Optimization is better understood as selective evidence-based simplification. Sometimes that simplification involves configuration tuning, sometimes it involves redesigning execution paths, and sometimes it involves removing unnecessary abstraction rather than adding more of it. This interpretation aligns with empirical studies that found many performance gains to be attainable through relatively contained changes when teams know where to look [5]. It also explains why testing and observability recur so frequently across the corpus. Both disciplines help teams decide where not to optimize blindly.

The authorial methodology has practical implications for software engineering education and professional work. For educators, it provides a way to teach web development as a method rather than as a stack checklist. Students often learn Spring, REST, browser scripting, and testing frameworks separately, yet struggle to reason about their interaction in one deployment pipeline. For practitioners, the framework offers a compact decision order: align the layers, verify early, instrument before guessing, optimize selectively, and evaluate long-term evolution cost. This order is not radical. Its value lies in making scattered evidence actionable.

There are important limitations. Because this article does not present a primary experiment, the proposed methodology should not be interpreted as universally validated under all project conditions. Different domains, scale levels, traffic profiles, and regulatory constraints may change the relative importance of testing depth, GUI automation, or runtime instrumentation. Rapidly evolving frameworks and deployment environments also mean that no methodology remains effective if it hardens into dogma. What should remain stable is not any specific tool preference, but the principle of cross-layer evidence. The scenario analysis in Table 5 shows how the phase order adapts to project context, but it is analytical and does not replace validation on a live project.

Future research can strengthen this work in at least three directions. First, the methodology could be tested on a live application that combines Java and JavaScript, with before-and-after measurements across latency, resource use, test failure rate, and maintenance effort. Second, repository-mining studies could evaluate whether projects that approximate the proposed cycle show lower defect recurrence or more stable optimization histories. Third, educational experiments could

examine whether students trained with a lifecycle-oriented approach produce more maintainable web solutions than those trained through isolated topic modules. These steps would move the present article from synthesis toward validation. Even so, the present discussion already makes one conclusion difficult to avoid: in modern web systems, optimization detached from methodology becomes expensive, and methodology that ignores optimization becomes incomplete.

6. CONCLUSIONS

This article has argued that web applications based on Java and JavaScript should be developed and optimized through a unified methodology rather than through disconnected technical interventions. By synthesizing twenty verified English-language studies, the paper identified several recurring principles. Cross-layer reasoning is more useful than layer isolation. Testing is foundational for trustworthy optimization. Runtime observability is essential for production-quality refinement. Maintainability considerations should remain visible during every serious optimization step.

The study did not conduct a primary empirical experiment, but it carried out a structured secondary empirical synthesis of prior findings and translated those findings into the Integrated Dual-Layer Optimization Cycle. That methodology constitutes the main practical contribution of the article. It is designed to help teams connect architecture, verification, instrumentation, targeted optimization, and evolutionary control within one engineering rhythm. The scientific value of the article lies in methodological integration. The practical value lies in redefining system optimization as a continuous state-space alignment between backend constraints and frontend execution environments, an alignment that begins with design and continues after deployment.

Future research should validate the proposed cycle in real projects and test how it behaves under different stack variants, team sizes, and delivery environments. Until then, the strongest conclusion supported by the literature is clear enough: the most effective way to improve web applications built with Java and JavaScript is not to optimize later and locally, but to build with optimization in mind from the start.

7. REFERENCES

- [1] Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 1-12). ACM. <https://doi.org/10.1145/1806596.1806598>
- [2] Sun, K., & Ryu, S. (2017). Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys*, 50(4), Article 59. <https://doi.org/10.1145/3106741>
- [3] Artzi, S., Dolby, J., Jensen, S. H., Möller, A., & Tip, F. (2011). A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 571-580). ACM. <https://doi.org/10.1145/1985793.1985871>
- [4] Doğan, S., Betin-Can, A., & Garousi, V. (2014). Web application testing: A systematic literature review. *Journal of Systems and Software*, 91, 174-201. <https://doi.org/10.1016/j.jss.2014.01.010>
- [5] Selakovic, M., & Pradel, M. (2016). Performance issues



- and optimizations in JavaScript: An empirical study. In Proceedings of the 38th International Conference on Software Engineering (pp. 61-72). ACM. <https://doi.org/10.1145/2884781.2884829>
- [6] Jensen, S. H., Møller, A., & Thiemann, P. (2009). Type analysis for JavaScript. In Static Analysis (pp. 238-255). Springer. https://doi.org/10.1007/978-3-642-03237-0_17
- [7] Madsen, M., Livshits, B., & Fanning, M. (2013). Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (pp. 499-509). ACM. <https://doi.org/10.1145/2491411.2491417>
- [8] Balsam, S., & Mishra, D. (2025). Web application testing: Challenges and opportunities. Journal of Systems and Software, 219, Article 112186. <https://doi.org/10.1016/j.jss.2024.112186>
- [9] Di Meglio, S., Starace, L. L. L., Pontillo, V., Opdebeeck, R., De Roover, C., & Di Martino, S. (2026). Investigating the adoption and maintenance of web GUI testing: Insights from GitHub repositories. Information and Software Technology, 189, Article 107928. <https://doi.org/10.1016/j.infsof.2025.107928>
- [10] Noetzold, D., Rossetto, A. G. de M., Silva, L. A., Crocker, P., & Leithardt, V. R. Q. (2024). JVM optimization: An empirical analysis of JVM configurations for enhanced web application performance. SoftwareX, 28, Article 101933. <https://doi.org/10.1016/j.softx.2024.101933>
- [11] Van Riet, J., Malavolta, I., & Ghaleb, T. A. (2023). Optimize along the way: An industrial case study on web performance. Journal of Systems and Software, 198, Article 111593. <https://doi.org/10.1016/j.jss.2022.111593>
- [12] Shatnawi, A., Rima, B., Alshara, Z., Darbord, G., Seriai, A.-D., & Moreau, P.-E. (2026). Observability of web applications: An industrial case study. Journal of Systems and Software, 225, Article 112886. <https://doi.org/10.1016/j.jss.2026.112886>
- [13] Wang, Z., Bu, D., Wang, N., Yu, S., Gou, S., & Sun, A. (2023). An empirical study on bugs in JavaScript engines. Information and Software Technology, 155, Article 107105. <https://doi.org/10.1016/j.infsof.2022.107105>
- [14] Jiang, Z., Zhong, H., & Meng, N. (2021). Investigating and recommending co-changed entities for JavaScript programs. Journal of Systems and Software, 180, Article 111027. <https://doi.org/10.1016/j.jss.2021.111027>
- [15] Rack, J., & Staicu, C.-A. (2023). Jack-in-the-box: An empirical study of JavaScript bundling on the web and its security implications. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (pp. 3198-3212). ACM. <https://doi.org/10.1145/3576915.3623140>
- [16] Yan, Y., Tu, T., Zhao, L., Zhou, Y., & Wang, W. (2021). Understanding the performance of WebAssembly applications. In Proceedings of the 21st ACM Internet Measurement Conference (pp. 533-549). ACM. <https://doi.org/10.1145/3487552.3487827>
- [17] Leitner, P., & Bezemer, C.-P. (2017). An exploratory study of the state of practice of performance testing in Java-based open source projects. In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (pp. 373-384). ACM. <https://doi.org/10.1145/3030207.3030213>
- [18] Imran, M., Cortellessa, V., Di Ruscio, D., Rubei, R., & Traini, L. (2024). An empirical study on code coverage of performance testing. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 150-159). ACM. <https://doi.org/10.1145/3661167.3661196>
- [19] Liu, X.-J., Yu, P., & Ma, X.-X. (2024). An empirical study on automated test generation tools for Java: Effectiveness and challenges. Journal of Computer Science and Technology, 39(3), 715-736. <https://doi.org/10.1007/s11390-023-1935-5>
- [20] Spadini, D., Aniche, M., Bruntink, M., & Bacchelli, A. (2019). Mock objects for testing Java systems: Why and how developers use them, and how they evolve. Empirical Software Engineering, 24, 1461-1498. <https://doi.org/10.1007/s10664-018-9663-0>