UDC 004.415:004.051:004.6
DOI https://doi.org/10.32782/2663-5941/2026.1.2/08

Hryshchenko A.I.
https://orcid.org/0009-0007-1191-7948
Lowe's Companies, Inc., NC, USA

PREVENTION OF PERFORMANCE REGRESSIONS IN WEB SYSTEMS AT THE IMPLEMENTATION AND OPERATION STAGES

The article addresses the problem of performance regressions in web systems that arise as a result of the evolution of program code, architectural decisions, and configurations at the implementation and operation stages. The distinction between performance regression as a version-dependent engineering defect and performance degradation as an operational phenomenon that manifests over time under the influence of external factors is demonstrated. The feasibility of a version-oriented approach is substantiated, within which performance control is carried out by comparing vectors of metrics between successive system versions using baselines and acceptable deviation thresholds. A version-oriented method for preventing performance regressions is proposed; it is integrated into CI/CD pipelines and provides for automatic identification of the set of modified components, execution of localized load-testing scenarios, evaluation of metric deviations, decision-making on build stability (pass/fail/rollback), and updating of baselines after stabilization. The theoretical justification of the method includes conditions for preserving the completeness of control under a controlled environment and the optimality of selective control according to the "time–completeness" criterion. The practical implementation is presented in the form of a procedure for applying the method in CI/CD using a component-to-test-scenario mapping matrix and repeated test runs to reduce the impact of measurement noise. Experimental validation on a web application with a three-tier architecture (Node.js/React/PostgreSQL) in a containerized environment with Jenkins, Prometheus, and Grafana confirmed that version-oriented control ensures complete detection of regressions and reduces testing time by 6–8 times compared to full regression testing, while decreasing the load on CI/CD infrastructure. The obtained results indicate the technological applicability of the method for deployment in production environments and form a basis for further development through adaptive metric analysis and regression forecasting.

Keywords: *web systems, performance, performance regression, baseline, CI/CD, DevOps, load testing, monitoring, performance metrics.*

Formulation of the problem. Modern web systems are characterized by a high pace of evolution, continuous software code updates, and complex multi-layer architectures. The adoption of continuous integration and delivery practices, as well as microservice and modular approaches to client-side and server-side development, increases development flexibility but simultaneously complicates the maintenance of stable system performance throughout its evolution. Under these conditions, even minor implementation or configuration changes may lead to the degradation of operational characteristics, manifesting as performance regressions.

Performance regressions in web systems represent a distinct class of engineering defects that, unlike functional errors, do not violate the correctness of program logic but significantly affect the efficiency, stability, and scalability of software systems. Such defects are often detected at later stages of integration or during real-world operation, which complicates their localization and remediation.

It is important to distinguish between performance degradation and performance regression. Performance degradation reflects a gradual or situational decline in system efficiency caused by external or cumulative factors and is not necessarily associated with software code changes. Performance regression, in contrast, results from modifications to implementation, architecture, or configuration and manifests as a deterioration of performance indicators between successive system versions. This distinction makes it possible to treat performance regression as a measurable, version-dependent property suitable for engineering analysis.

In most practical solutions, performance regressions are addressed fragmentarily – either as a load

testing task or as an operational monitoring problem. The lack of integration between implementation, testing, and operational stages leads to a reactive detection of degradations and fails to ensure systematic performance quality control.

Therefore, the development of engineering methods for preventing performance regressions in web systems, focused on the implementation and operational stages, is a relevant research problem. Such an approach implies continuous performance control as a property of an evolving system throughout its entire software life cycle.

Analysis of recent research and publications. The problem of detecting and preventing performance regressions in web systems has been actively studied over the past decade in the context of the development of continuous integration and delivery (CI/CD) methods. Contemporary publications demonstrate an evolution from traditional testing techniques toward comprehensive automated systems focused on data analytics, machine learning, and continuous monitoring.

One of the most recent directions is the deployment of high-precision monitoring systems in production environments. In [1], an approach is presented that enables the detection of even minimal performance changes (below 1%) in large-scale services with use of internal telemetry. A similar idea is further developed in [2], where the combination of local measurements with architectural models is proposed to predict degradations at early stages of the software life cycle.

The development of statistical methods for analyzing time series of testing results is demonstrated in [3], which proposes algorithms for automatic detection of performance changes in CI/CD pipelines. Similar principles are applied in [4], where the use of change point detection methods made it possible to reduce the number of false positives and shorten the time required to respond to system degradations. The use of machine learning models for diagnosing regressions without the need for prior data labeling is considered in [5], which describes an anomaly-based learning approach capable of identifying rare or difficult-to-analyze degradations.

Researchers also invest significant effort in the creation of open datasets suitable for training and validating new performance control methods. In [6], a public dataset is described that contains thousands of time series and regression alerts obtained during the testing of the Mozilla Firefox browser. Such resources provide the research community with real-world examples for developing degradation detection algorithms and automating performance analysis.

The effectiveness of regression testing in CI/CD environments is examined in [7], where it is shown that optimizing test execution order and applying incremental checks can reduce total testing time by more than half. In [8], a risk-oriented approach is proposed that determines testing priorities based on the estimated probability of degradation, thereby minimizing resource consumption while maintaining control quality.

Further progress has been made in benchmarking optimization within CI/CD pipelines. In [9], a conceptual cloud-based platform is proposed that combines frequent but low-overhead testing with automatic result evaluation. This approach helps reduce operational overhead and increases the practicality of regular performance analysis in industrial environments.

Modern studies pay particular attention to the integration of automated control systems directly into continuous development processes. In [10], an example of integrating a change detection service into GitHub Actions is demonstrated, enabling the automatic identification of regressions caused even by changes in the execution environment, such as updates to the operating system kernel version.

The overall analysis of the literature shows that the research community is gradually shifting from reactive to preventive performance control strategies. While earlier approaches focused on detecting already manifested degradations, contemporary solutions integrate prediction, adaptive learning, and the formation of performance baselines. Of particular relevance is the version-oriented approach, which enables the analysis of performance changes in the context of code and architecture evolution, providing a foundation for continuous engineering control of web system quality.

Task statement. The objective of this study is to develop and analyze engineering methods for detecting and preventing performance regressions in web systems at the stages of their implementation and operation through the integration of automated testing, analysis of software code changes, and the use of operational data. To achieve this objective, the article examines the causes of performance regressions, analyzes architectural and implementation-related factors of performance degradation, and formulates an integrated engineering approach to ensuring stable performance of web systems under conditions of continuous change.

Outline of the main material of the study. During the evolution of web systems, software performance may change as a result of modifications to program

code, architectural decisions, or execution environment configurations. Unlike initial design shortcomings, performance regressions arise in already functionally correct systems and manifest themselves as a deterioration of operational characteristics between different software versions.

In the context of software engineering, performance regressions should be considered as defects that arise at the intersection of the implementation, integration, and operation stages of a system. Their occurrence is driven by continuous changes in program code and architecture in the absence of systematic control over operational characteristics during system evolution. This substantiates the need for an engineering approach aimed not only at detecting but also at preventing performance regressions throughout the web system life cycle.

Figure 1 illustrates the life cycle of a web system from the perspective of the emergence and control of performance regressions. During the implementa-

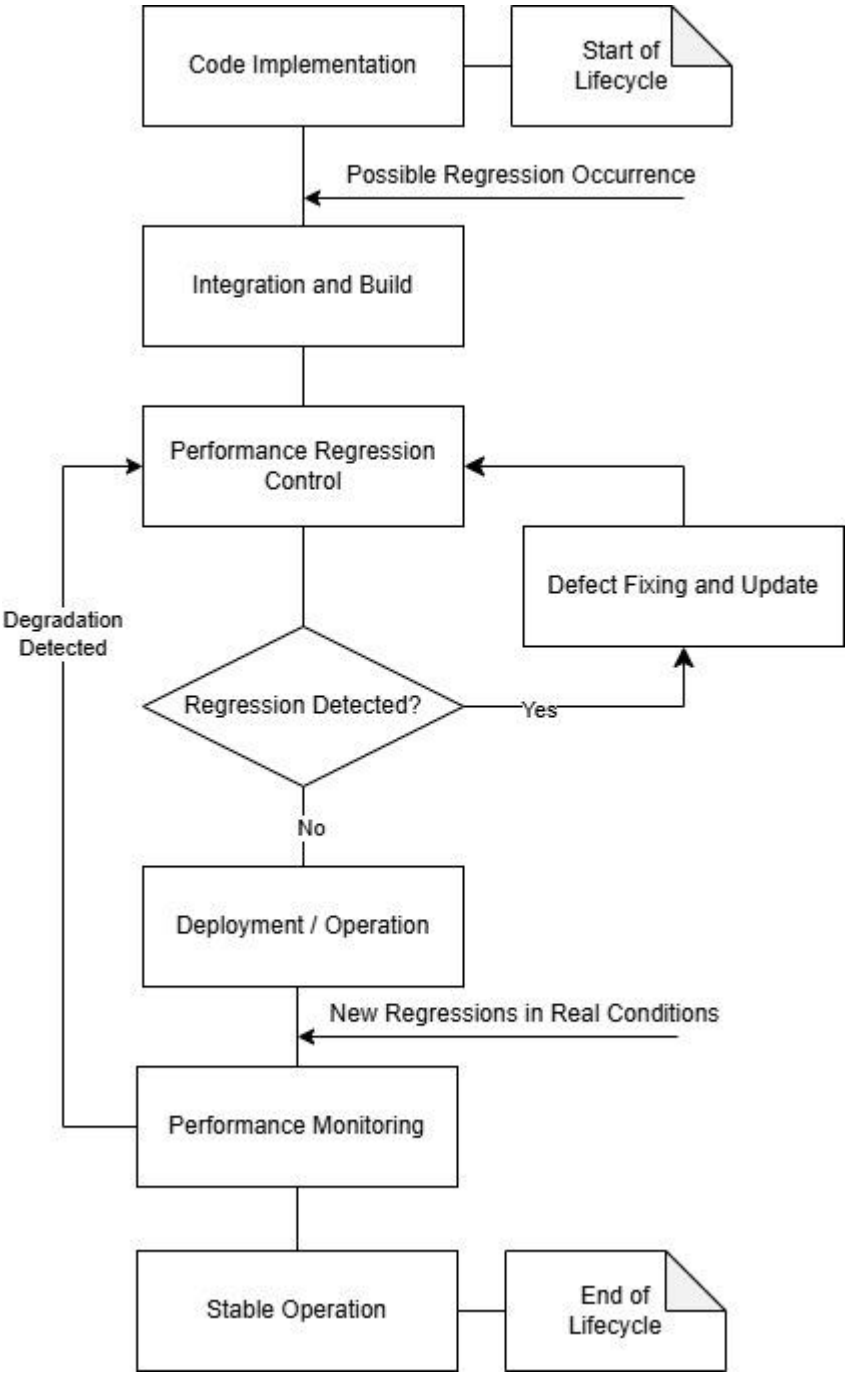

Fig. 1. Position of performance regressions in the life cycle of a web system

tion and modification of program code, a new system version is formed, which then passes through the integration and build stage. Subsequent performance control is carried out by comparing the metrics of the current version with baseline values established for previous releases, enabling decisions on deployment, rollback to a previous version, or elimination of identified defects. During operation, the system is subject to continuous performance monitoring, the results of which provide feedback and are used to control subsequent versions, ensuring a closed loop for maintaining performance stability.

For engineering analysis of performance regressions, it is appropriate to use a version-oriented approach, within which the system state is considered as a set of implementation, architectural, and operational characteristics associated with a specific software version. Within this approach, a regression is defined as a deviation of the performance indicators of version v_{n+1} from the corresponding indicators of version v_n under unchanged or controlled execution conditions.

A key element of this approach is the formation of performance baselines that represent acceptable bounds of variation in operational characteristics between successive versions. The use of baselines makes it possible to distinguish random metric fluctuations from systematic degradations caused by changes in program code or architecture. Such a model provides a formalized foundation for automated regression analysis within continuous integration and delivery processes.

An important advantage of the version-oriented approach is the ability to localize regressions by correlating performance changes with specific implementation or architectural modifications. This creates prerequisites for a transition from reactive problem fixing to engineering-based management of web system performance quality during system evolution.

Figure 2 illustrates the generalized dynamics of changes in the performance of a web system across successive versions. The example shows that after stable versions v_1–v_2, a significant degradation of performance metrics (a regression) occurs in version v_3, which requires optimization. After the defects are eliminated, a new performance baseline is established for version v_4. This approach makes it possible to clearly demonstrate the mechanism for controlling performance stability within the version-oriented method.

Throughout the evolution of web systems, maintaining stable performance is one of the key aspects

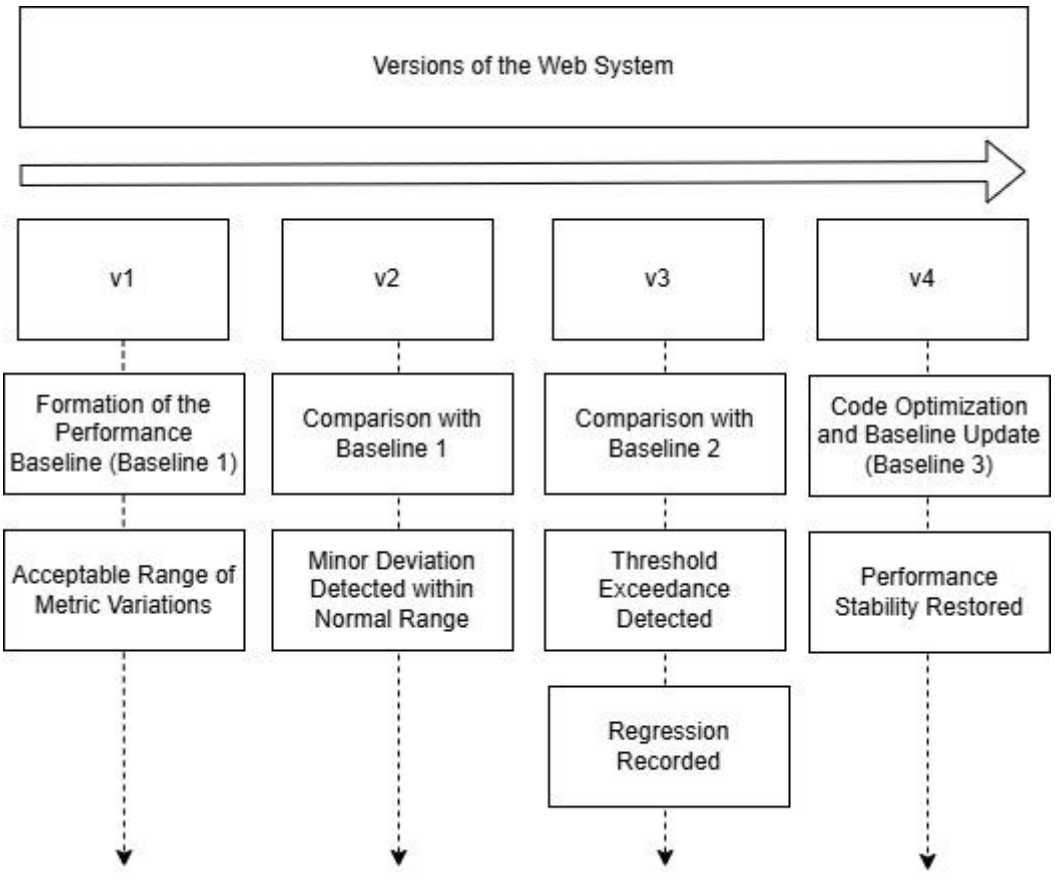

Fig. 2. Comparison of system versions and performance baselines

of software quality assurance. Unlike classical load testing methods, which are predominantly reactive in nature, the proposed version-oriented method provides proactive control by comparing performance characteristics between successive system versions. Its distinctive feature is the formalization of the control process in the form of a cycle that is integrated into the software development and operation life cycle.

The version-oriented method is based on the idea that the performance state of a system is determined not by absolute values, but by relative changes between versions. Each version of a web system v_i is characterized by a set of measurable performance indicators:

$$M_i = \{t_{resp}, t_{proc}, CPU, RAM, IO, RPS\},$$

where t_{resp} – the average system response time;
t_{proc} – the average server-side request processing time;
CPU – the average central processing unit utilization during the test;
RAM – denotes main memory usage;
IO – represents the intensity of input/output operations;
RPS – the number of processed requests per second.

For a stable version, a performance baseline is established as a vector of reference values B_i, against which the metrics of subsequent versions v_{i+1} are compared. During testing of a new version, the metric vector M_{i+1} is measured, and the vector of component-wise deviations is defined as:

$$\Delta_{i \to i+1} = \frac{M_{i+1} - B_i}{B_i} \times 100\%.$$

The system is considered stable if, for each metric, the deviation does not exceed the corresponding threshold. If the threshold is exceeded for any of the indicators, a performance regression is recorded and localized to the components that changed between the versions.

Theoretical justification of the method. Let S denote the set of all system components, and let $Sc \subseteq S$ be the set of components modified in the current version. For each component $s_i \in S$, there exists a mapping that describes the impact of code changes on performance metrics:

$$f_i : C_i \to M,$$

where C_i denotes the space of code states of a component, and M is the space of measured performance characteristics. It is assumed that the functions f_i are locally Lipschitz, that is, there exist constants $L_i > 0$ such that:

$$f_i(C_i') - f_i(C_i'') \le L_i C_i' - C_i''.$$

This condition reflects an engineering stability property: small changes in the code lead to proportionally small changes in performance, which makes it possible to restrict control to the subset Sc without losing the completeness of degradation detection.

Theorem 1 (On preserving the completeness of performance control under controlled environment conditions).

Let S be the set of all system components, and let $S_c \subseteq S$ be the subset of modified components. If, for each $s_i \in S$, the mapping $f_i : C_i \to M$ is locally Lipschitz, and the execution conditions and interaction interfaces of the components remain unchanged, then testing only on the set S_c ensures the completeness of performance control:

$$P_{miss}(S_c) = 0.$$

Proof. For all $s_j \notin S_c$, the condition $||C_j' - C_j''|| = 0 \Rightarrow ||f_j(C_j') - f_j(C_j'')|| = 0$ holds. Therefore, the contribution of unchanged components to the performance metric vector is invariant under fixed conditions. Since performance regressions under fixed conditions are caused by changes in S_c, controlling these components guarantees the completeness of degradation detection within the adopted assumptions. Thus, testing only the modified components does not reduce the completeness of control, while significantly reducing the testing effort.

In cases where execution conditions or environment configurations change, or when component modifications lead to indirect effects on adjacent components, the completeness of control may require extending the set S_c. In such scenarios, the version-oriented method can be complemented by operational monitoring or periodic global performance control.

Theorem 2 (On the optimality of version-oriented control under the time–completeness criterion).

Let $T(S)$ denote the total time required to control all system components, $T(S_c)$ the time required to control the modified components, $r = |S_c|/|S|$ the proportion of modified components, and $P_d(S_c)$ the probability of detecting regressions when controlling the component set S_c. Under the controlled environment conditions defined in Theorem 1, we define the control efficiency as

$$E(S_c) = P_d(S_c) - \lambda \frac{T(S_c)}{T(S)},$$

where $\lambda \in (0,1)$ is a weighting coefficient for time costs.

Then, the optimal control region that maximizes $E(S_c)$ coincides with the set of modified components S_Δ:

$$S_c^* = S_\Delta,\ E(S_c^*) = 1 - \lambda r.$$

Proof. For full control, $P_d(S) = 1$ and $T(S_c) = T(S)$, therefore $E(S) = 1 - \lambda$. For a set $S_c \supseteq S_\Delta$, the detection probability is also equal to 1, while the control time decreases to $T(S_c) = rT(S)$. Then, $E(S_c) = 1 - \lambda r$, which is maximized for the minimal value of r, i.e., when $S_c = S_\Delta$. Thus, controlling only the components that changed between versions is optimal: it preserves completeness while minimizing time costs.

Implementation of the version-oriented method in the CI/CD process includes:

1. Automatic identification of changed components S_c after each commit or merge.
2. Execution of localized performance tests only for the scope S_c.
3. Comparison of new metrics with the baseline B_i and detection of regressions.
4. Triggering a rollback mechanism when allowable deviations ε_i are exceeded.
5. Automatic baseline updates after system stabilization.

This approach minimizes testing duration while preserving completeness of degradation detection and enables continuous control of performance stability throughout the entire web system life cycle.

The version-oriented method for preventing performance regressions formalizes the evolutionary control process as an iterative sequence:code change → selective verification → deviation analysis → baseline update.

Due to the theoretically proven properties of completeness and optimality, the method provides a balance between testing speed and reliability of regression detection, forming a basis for integrating automated performance control into DevOps and CI/CD practices.

The procedure for applying the proposed version-oriented method in CI/CD pipelines defines a sequence of actions that ensures practical implementation of performance regression control under conditions of continuous integration and delivery of software. The procedure is aimed at automated detection of degradations between successive versions of a web system with minimal time costs and without reducing the completeness of control.

At the first stage, identification of the modified system components is performed. For this purpose, after each commit or merge, the difference between the current and the previous versions of the source code is analyzed using version control mechanisms. Based on this analysis, the set of modified components S_c is formed, which determines the scope of subsequent performance control.

At the second stage, a correspondence between the modified components and test scenarios is established. For each component in the set S_c, a set of load-testing scenarios that affect its operational characteristics is determined. This correspondence is formed based on architectural knowledge of the system and the results of previous measurements, and it makes it possible to limit testing to only relevant scenarios.

The third stage involves executing selective performance tests. Load testing is performed only for scenarios associated with the components in the set S_c, which provides a significant reduction in the duration of the testing cycle compared to full regression testing.

At the fourth stage, performance metrics are collected and aggregated. During testing, key operational indicators are recorded (response time, request processing time, CPU and memory utilization, input/output operation intensity, and throughput). Based on the measurement results, the metric vector M_{i+1} is formed for the current system version.

The fifth stage consists in comparing the obtained indicators with the performance baseline B_i. Relative deviations of the metrics are calculated and their compliance with acceptable threshold values is verified. To reduce the impact of random fluctuations, test scenarios may be executed multiple times, and aggregated metric values are used for analysis.

At the sixth stage, a decision on version stability is made. If no threshold exceedances are detected, the version is considered stable and is allowed for further deployment. If a performance regression is detected, the procedure initiates appropriate actions, such as stopping the pipeline, rolling back to a previous version, or transferring the results to developers for elimination of the identified defects.

The final stage is updating the performance baseline. After regressions are eliminated and the stability of the new version is confirmed, its operational characteristics are fixed as a new reference for subsequent control iterations.

The selection of test scenarios is carried out based on a correspondence rule between system components and load scenarios. For this purpose, a coverage matrix of the form *Coverage: component → {test scenarios},* is formed, which determines which testing scenarios affect the performance of a particular component. This approach makes it possible to ensure

completeness of control when selectively testing only the modified components.

To reduce the impact of random performance fluctuations, each test scenario is executed k times, after which the median value of the metrics is used for analysis. A regression is recorded only in the case of a stable exceedance of threshold deviations, which makes it possible to distinguish systemic degradations from measurement noise.

In order to practically validate the effectiveness of the developed version-oriented method for preventing performance regressions, an experimental study was conducted. Its objective was to evaluate two key indicators – regression detection completeness and time efficiency – in comparison with traditional full regression testing. The study was carried out under conditions close to real DevOps processes of continuous integration and delivery of software.

The experiments were performed in an environment that reproduced a typical CI/CD cycle of a web system. Testing was conducted on a server with the following specifications: an Intel Xeon Silver 4210 processor (20 threads, 2.2 GHz), 32 GB of RAM, and the Ubuntu Server 22.04 LTS operating system. All system components were deployed in Docker Compose containers, which ensured reproducibility and stability of configurations. Build and test automation was implemented using Jenkins, with plugins for monitoring changes in the Git repository. Operational metric monitoring was carried out via Prometheus, and result visualization was provided through Grafana. Apache JMeter was used for load testing.

The test object was a web application of the *Performance Monitoring Dashboard* type – a system for collecting, analyzing, and visualizing telemetry data from server infrastructure. The application was implemented using a three-tier *client–server–database* architecture, which provides a clear separation of responsibilities among system components. The server-side (backend) was developed on the Node.js platform using the Express.js framework, which implements both REST and WebSocket APIs for client interaction. The client-side (frontend) was built with React.js and uses the D3.js and Chart.js libraries to construct dynamic interactive charts and to visualize data in real time. The data storage subsystem is based on PostgreSQL 15, which supports query caching and operation logging to improve access efficiency. Asynchronous message exchange between services is implemented using RabbitMQ, ensuring reliable communication and system scalability.

The typical application load corresponds to approximately 50–80 concurrent users with an average throughput of about 1000 HTTP requests per minute. The system architecture was deliberately chosen to reflect the characteristics of industrial web systems, including multi-layer business logic, intensive interaction with the database, and sensitivity to performance changes at both the server component level and the client interface level.

For the study, four successive versions of the system were prepared – from a baseline stable version to modified versions with additional functionality (Table 1). In each version, a specific part of the code was changed, which made it possible to evaluate the method's response to different types of modifications.

For each version, a performance baseline was created, containing averaged values of response time, CPU utilization, memory usage, input/output (I/O) throughput, and the number of processed requests per second (RPS). These indicators served as reference values for subsequent versions.

To simulate real operating conditions, five test scenarios were developed and executed in JMeter:

1. Login / Logout – verification of user authentication and session creation;
2. Data Query – execution of database queries with varying data volumes;
3. Data Update – updating and inserting records into database tables;
4. Dashboard Render – rendering of the analytics dashboard with API requests and chart generation;
5. Concurrent Load – simulation of 50 concurrent users issuing combined requests.

Each scenario was executed for 10 minutes, with a gradual increase in load up to 1000 requests per minute.

Table 1

Evolution of web application versions

Version	Modified modules	Share of modified code	Nature of changes
v_1	Baseline stable version	–	–
v_2	Database query module	7%	Refactoring of SQL queries and index modification
v_3	API and logging module	12%	Introduction of query caching and asynchronous logging
v_4	Client-side interface	15%	Migration from AJAX to WebSockets for chart updates

The testing process consisted of the following stages:

1. Building and deploying a new system version using a Jenkins pipeline.
2. Identification of modified components based on Git diff results between versions.
3. Selection of the testing mode: full testing of all scenarios; version-oriented testing of only the modified modules.
4. Performance measurement: collection of CPU, RAM, RPS, response time, and API latency metrics.
5. Comparison with baseline values: calculation of deviations as percentages of the reference values.
6. Regression detection: automatic generation of reports in Jenkins and Prometheus AlertManager when deviation thresholds are exceeded.
7. Baseline update: after optimization and repeated testing, new stable indicators were fixed as a new baseline.

The experimental results showed that applying the version-oriented method made it possible to significantly reduce the time required for performance control without loss of accuracy. On average, full testing took 42 minutes, whereas version-oriented testing required only 5–6 minutes, that is, approximately seven to eight times less.

During the experiment, seven regressions were identified, in particular: in version v_2, an increase in the average database query execution time by 6.4%; in version v_3, an increase in CPU load by 9.7% due to excessive logging; in version v_4, an increase in the average chart rendering time by 12.1% after the transition to WebSockets; in version v_2, an increase in memory consumption by 5.1% in the Data Update scenario; in version v_3, a decrease in RPS by 7.8% in the Concurrent Load scenario; in version v_4, an increase in the average WebSocket message latency by 8.6%; in versions v_3–v_4, an increase in I/O wait time by 6.9% during execution of the Dashboard Render scenario.

In all cases, regressions were detected by both methods, which confirmed the completeness of control. After optimization of the code changes, the metrics returned to baseline ranges, and the updated values were fixed as new reference baselines for subsequent versions. The obtained results confirmed the main provisions of the theoretical model. First, controlling only the modified components ensured complete regression detection, meaning that all performance degradations were identified without omissions. Second, the method demonstrated a linear reduction in testing time directly proportional to the share of modified code. When 10–15% of components were changed, the average reduction in control time was 80–90%, while accuracy remained at 100%.

Additionally, a reduction in CI/CD infrastructure load was observed: average CPU utilization during tests decreased by 60%, and the average time between commit and release was reduced from 48 to 9 minutes. These results indicate a significant increase in the efficiency of the development process without compromising control quality.

For practical deployment of the version-oriented method, it is advisable to store performance baselines as versioned artifacts (e.g., Git LFS, a database, or Prometheus metric snapshots). Baseline versioning should be synchronized with software code versions. To reduce the number of false positives, it is recommended to perform system warm-up before testing, fix execution resources, and apply nightly test runs under stable load conditions.

Conclusions. This paper addresses the problem of preventing performance regressions in web systems that arise during the implementation and operation stages. The nature of performance regressions is analyzed as a specific class of engineering defects that differ from functional errors in that they do not violate the logic of program execution but lead to a decrease in system efficiency and stability. A version-oriented performance control method is proposed, which is based on forming performance baselines for each software version and comparing vectors of performance metrics between successive versions. This approach enables integration of regression control into continuous integration and delivery (CI/CD) processes, providing automated detection of degradations during system evolution.

Experimental validation demonstrated that the proposed version-oriented method is practically effective and technologically compatible with modern DevOps processes. Its application makes it possible to reduce regression testing time by a factor of 6–8, decrease the load on CI/CD servers by more than 50%, detect all performance regressions without loss of accuracy, and ensure continuous control of web system performance stability throughout evolution.

The obtained results confirm that the method can be deployed in real production environments to build automated monitoring systems and prevent performance degradations in dynamic web applications. Further research should focus on extending the method by incorporating adaptive metric analysis using machine learning for regression prediction, as well as on integrating the approach with cloud-based monitoring systems to ensure scalability and autonomy of performance control in large distributed web systems.

Bibliography:

1. Yoon D. Y., Wang Y., Yu M., Kim H. FBDetect: Catching Tiny Performance Regressions at Hyperscale through In-Production Monitoring, *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP '24)*, Austin, TX, USA, 2024. DOI: https://doi.org/10.1145/3694715.3695977 (date of access: 10.01.2026).
2. Liao L., Eismann S., Li H., Rohr M., Reussner R. Early Detection of Performance Regressions by Bridging Local Performance Data and Architectural Models, *arXiv preprint*, 2024. URL: https://arxiv.org/abs/2408.08148 (date of access: 10.01.2026).
3. Fleming M., Kołaczkowski P., Kumar I., Trubiani C. Hunter: Using Change Point Detection to Hunt for Performance Regressions, *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '23)*, 2023. URL: https://arxiv.org/abs/2301.03034 (date of access: 10.01.2026).
4. Daly D., Brown W., Ingo H., O'Leary J., Bradford D. Change Point Detection in Software Performance Testing, *arXiv preprint*, 2020. DOI: https://doi.org/10.48550/arXiv.2003.00584 (date of access: 10.01.2026).
5. Alam M., Gottschlich J., Tatbul N., Turek J., Mattson T., Muzahid A. A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions, *arXiv preprint*, 2017. URL: https://arxiv.org/abs/1709.07536 (date of access: 10.01.2026).
6. Besbes M. B., Costa D. E., Mujahid S., Mierzwinski G., Castelluccio M. A Dataset of Performance Measurements and Alerts from Mozilla (Data Artifact), *arXiv preprint*, 2025. DOI: https://doi.org/10.48550/arXiv.2503.16332 (date of access: 10.01.2026).
7. Lukavenko D., Delembovskyi M. Efficiency of Regression Testing Strategies in CI/CD Environments, *CEUR Workshop Proceedings*, 2023. URL: https://ceur-ws.org/Vol-3896/paper33.pdf (date of access: 10.01.2026).
8. Huang P., Ma X., Shen D., Zhou Y. Performance Regression Testing Target Prioritization via Performance Risk Analysis, *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Hyderabad, India, 2014. P. 60–71. DOI: https://doi.org/10.1145/2568225.2568232 (date of access: 10.01.2026).
9. Japke N., Koch S., Lukasczyk H., Bermbach D. Towards an Optimized Benchmarking Platform for CI/CD Pipelines, *Proceedings of the 2025 IEEE International Conference on Cloud Engineering (IC2E)*, 2025. URL: https://arxiv.org/abs/2510.18640 (date of access: 10.01.2026).
10. Yang S., Reichelt D. G., Ingo H., Hasselbring W. Detection of Performance Changes in MooBench Results Using Nyrkiö on GitHub Actions, *arXiv preprint*, 2025. DOI: https://doi.org/10.48550/arXiv.2510.11310 (date of access: 10.01.2026).

Грищенко А.І. ЗАПОБІГАННЯ РЕГРЕСІЯМ ПРОДУКТИВНОСТІ ВЕБ-СИСТЕМ НА ЕТАПАХ РЕАЛІЗАЦІЇ ТА ЕКСПЛУАТАЦІЇ

У статті розглянуто проблему регресій продуктивності веб-систем, що виникають унаслідок еволюції програмного коду, архітектурних рішень і конфігурацій на етапах реалізації та експлуатації. Показано відмінність регресії продуктивності як версіє-залежного інженерного дефекту від деградації продуктивності як експлуатаційного явища, що проявляється у часі під впливом зовнішніх факторів. Обґрунтовано доцільність версіє-орієнтованого підходу, у межах якого контроль продуктивності здійснюється шляхом порівняння векторів метрик між послідовними версіями системи з використанням базових ліній (baseline) та порогів допустимих відхилень. Запропоновано версіє-орієнтований метод запобігання регресіям продуктивності, який інтегрується у CI/CD-конвеєри та передбачає автоматичну ідентифікацію множини змінених компонентів, запуск локалізованих навантажувальних сценаріїв, оцінювання відхилень метрик, ухвалення рішення щодо стабільності збірки (pass/fail/rollback) і оновлення базових ліній після стабілізації. Теоретичне обґрунтування методу включає умови збереження повноти контролю за контрольованого середовища та оптимальність вибіркового контролю за критерієм «час–повнота». Практичну реалізацію подано у вигляді процедури застосування методу у CI/CD із використанням матриці відповідності «компонент → тестовий сценарій» та повторних прогонів для зменшення впливу вимірювального шуму. Експериментальна перевірка на веб-застосунку трирівневої архітектури (Node.js/React/PostgreSQL) у контейнеризованому середовищі з Jenkins, Prometheus і Grafana підтвердила, що версіє-орієнтований контроль забезпечує повноту виявлення регресій і скорочує час тестування у 6–8 разів порівняно з повним регресійним тестуванням, знижуючи навантаження на CI/CD-інфраструктуру. Отримані результати свідчать про технологічну придатність методу для впровадження у виробничих середовищах та формують основу для подальшого розвитку шляхом адаптивного аналізу метрик і прогнозування регресій.

Ключові слова: *веб-системи, продуктивність, регресія продуктивності, baseline, CI/CD, DevOps, навантажувальне тестування, моніторинг, метрики продуктивності.*

Дата першого надходження статті до видання: 16.01.2026
Дата прийняття статті до друку після рецензування: 12.02.2026
Дата публікації (оприлюднення) статті: 08.04.2026