Managing Technical Debt While Migrating Long-Standing Legacy Code to A Modern Tech Stack

[1] Nizamutdinov Ilnar Rakipovich
[1] Full stack developer Belgrade, Serbia

Received: 27th Nov 2025 | Received Revised Version: 24th Dec 2025 | Accepted: 17th Jan 2026 | Published: 31st Jan 2026

Volume 08 Issue 01 2026 | 10.37547/tajiir/Volume08Issue01-17

Abstract

The issue of managing technical debt (TD) in large-scale organizations with long-operated legacy systems in 2024–2025 is becoming a key strategic factor, since TD is being transformed into the dominant source of operational risks and loss of market competitiveness. The refusal to pursue consistent modernization leads to the accumulation of TD caused by poor software code quality and inert, obsolete architecture, as a result of which up to 42% of developers' working time is spent on rework and defect remediation, which causes a critical slowdown in the Time to Market (TTM) metric and deterioration of the user experience (UX), evaluated by Core Web Vitals indicators, in particular Largest Contentful Paint (LCP) and Time to Interactive (TTI). Under these conditions, the aim of the study is formulated as the development of a holistic integrated approach to strategic TD management, within which incremental architectural migration methods are aligned with quantitatively measurable business outcomes. The empirical and theoretical basis of the study includes a systematic review of current academic publications in the IEEE, ACM, and Scopus databases, as well as an in-depth case study analysis of the migration of a large e-commerce platform to the ReactJS/TypeScript technology stack. The leading methodological foundation is the Strangler Fig Pattern (SFP), which is used for stepwise and controlled reduction of architectural risks during evolutionary system transformation. The results obtained show that the combination of incremental migration with preventive management of code debt through static typing (TypeScript) provides a statistically significant reduction in TTM and a substantial improvement in LCP and TTR metrics, which is in direct correlation with increased conversion and reduced total operating costs. The practical significance of the provisions presented lies in the formation of the concept of a managed digital core, the maintenance of which requires the annual allocation of about 15% of the IT budget to software-autonomous TD remediation, which creates a normative basis for long-term investment planning by CIOs and architectural teams.

Keywords: Technical debt, Legacy systems, Migration, Strangler Fig Pattern, ReactJS, TypeScript, Time to Market (TTM), LCP, DevOps, Architectural debt.

Cite This Article: Rakipovich, N. I. (2026). Managing Technical Debt While Migrating Long-Standing Legacy Code to A Modern Tech Stack. The American Journal of Interdisciplinary Innovations and Research, 8(01), 127–137. https://doi.org/10.37547/tajiir/Volume08Issue01-17

Introduction

Software, in its evolutionary characteristics, approximates a living system and is naturally subject to the phenomenon of ageing, in which technical debt accumulates over time as an inevitable result of continuous evolution and iterative development [6]. In the original understanding proposed by Ward Cunningham in 1992, technical debt was described as the need to expend additional effort to correct previously written code due to changes in the current understanding of the domain, requirements, and architectural decisions

[7]. In subsequent decades, the concept expanded significantly and ceased to be limited exclusively to the initial debt, having come to include architectural, code, infrastructural, and process components of debt that form a multidimensional space of risks for complex software ecosystems [8].

In the modern competitive environment, especially in the segment of large e-commerce and high-load digital platforms, the management of technical debt has definitively gone beyond a purely engineering problem and has transformed into a category of strategic business risk. Reporting data for 2024–2025 show that ignoring technical debt leads to a sharp increase in the cost of operation: this is manifested in the growth of penalties for non-compliance with regulatory requirements (compliance), an increase in cyber vulnerability due to the use of unsupported libraries and frameworks, as well as a decrease in the efficiency of the use of cloud infrastructure [1]. Legacy systems that support business-critical processes are often built on monolithic architecture, which is fundamentally poorly compatible with modern DevOps practices and complicates the implementation of continuous integration and continuous delivery (CI/CD) pipelines, which additionally reinforces the structural inertia of such solutions [10].

The relevance of the problem of legacy code migration is determined not only by the need for technological modernization, but also by the direct, quantitatively measurable impact of technical debt on the productivity of development teams and Time to Market (TTM) indicators. Low code quality, which is typical for systems with a long operational history and multiple layers of changes, leads to a situation in which developers are forced to spend up to 42% of their working time on rework and defect correction, and the time required to resolve issues in such code can increase by up to 124% compared to systems with a lower level of debt [3]. Under these conditions, technical debt functions as a direct barrier to organizational agility and to the speed of bringing new products and features to market, blocking the achievement of target TTM values and reinforcing business dependency on obsolete solutions [11]. In a strategic context, leading consulting companies consider a certain volume of technical debt as an inevitable price of innovation, while emphasizing the necessity of deliberate, managed allocation of resources to its remediation. Empirical data from global surveys indicate that organizations with a mature digital strategy seek to balance current technical debt obligations with investments in future development, systematically allocating about 15% of the total IT budget to the planned reduction of accumulated debt [4].

The degree of development of the topic is substantial in terms of the formalization of taxonomies and methods for detecting technical debt, which is reflected in particular in the materials of the International Conference on Technical Debt [13]. In the area of large-scale system migration, incremental approaches have gained wide recognition, among which a key place is occupied by the Strangler Fig Pattern (SFP), which makes it possible to gradually decommission monolithic functionality through the stepwise replacement of this functionality with modular components while minimizing operational risk and reducing the probability of critical failures [15]. At the same time, there remains a deficit of comprehensive studies that link, within a single model, the choice of a specific architectural strategy (SFP), the prevention of new code debt through the use of modern technology stacks (for example, ReactJS and TypeScript), and the quantitative impact of such decisions on key business metrics, including TTM and indicators of user experience quality measured through Core Web Vitals (LCP, TTR).

Under these conditions, the **aim** of the study is formulated as the development of an integrated approach to managing technical debt in the process of legacy system migration, oriented toward reducing architectural risks and achieving quantitatively fixed improvement of the main business and performance metrics. To achieve this aim, interrelated research tasks are consistently solved: an analysis of the modern taxonomy of technical debt is carried out with the substantiation of its financial impact on development processes, including losses of working time and an increase in defect frequency; an assessment of the architectural pattern is performed as a tool for managing architectural debt and risks during migration; the role of the ReactJS/TypeScript technology stack is determined as a mechanism for preventing code debt and increasing the long-term maintainability of the system; an empirical assessment is carried out of the impact of strategic migration on key metrics Time to Market (TTM), Largest Contentful Paint (LCP), and Time to Render (TTR) using the example of a large e-commerce platform.

The scientific novelty of the study consists in the formulation and empirical verification of a model for managing technical debt in which the incremental architectural pattern, the modern frontend stack, and the TTM and LCP metrics are integrated into a single concept at the level of business effect, which makes it possible to interpret architectural decisions as a direct driver of economic indicators.

The author hypothesis assumes that strategically planned incremental migration of long-standing legacy code using the Strangler Fig Pattern and the introduction of static typing (TypeScript) ensures a statistically significant reduction in Time to Market and a substantial improvement in Core Web Vitals indicators (LCP, TTR), which in turn is directly translated into the growth of key business indicators.

Materials and Methods

The study relies on a combined methodological approach integrating a systematic literature review in the format of a Systematic Mapping Study with an in-depth case study analysis of the migration of a large-scale production system. This combination provides, on the one hand, a high level of scientific rigor due to reliance on peer-reviewed sources and, on the other hand, the possibility of transferring theoretical conclusions into a real industrial context for their empirical validation.

The systematic review focuses on materials from leading academic databases (Scopus, Web of Science, IEEE Xplore, ACM Digital Library), as well as analytical reports of consulting agencies. The key selection criteria included explicit mention of technical debt, descriptions of legacy system migration strategies (with an emphasis on incremental approaches), the presence of TTM, LCP, and TTR metrics, and the use of modern technology stacks such as ReactJS and TypeScript in the context of large-scale development. The collected body of publications was systematized into thematic clusters reflecting different dimensions of the problem under study. The first cluster describes the economic and business impact of technical debt and includes quantitative studies demonstrating the relationship between code quality and financial indicators such as defect frequency, time to resolution, and the share of lost developer working time; this cluster also includes data on strategic budgeting for technical debt remediation and the allocation of corresponding costs. The second cluster is devoted to architectural migration patterns, with central attention given to the Strangler Fig Pattern (SFP), its practical implementations, and its role in minimizing architectural risks during evolutionary system transformation. The third cluster focuses on technologies for preventing technical debt and analyzes the contribution of static typing (TypeScript) and component-based frontend frameworks (ReactJS) to ensuring the long-term stability of architecture and simplifying refactoring. The fourth cluster brings together works that address performance and user experience quality metrics, as well as quantitative relationships between TTM, LCP, TTR, conversion indicators, and the rate of refusals in e-commerce.

For empirical validation of the hypothesis, a quantitative analysis of three groups of metrics is employed. The first group reflects organizational flexibility (Agility) and is represented by the Time to Market (TTM) metric, which measures the interval from idea formation to product deployment in production; a reduction in TTM is interpreted as a direct consequence of the reduction of architectural and code debt. The second group includes user experience (UX) quality metrics, primarily Largest Contentful Paint (LCP) and Time to Response (TTR), where LCP characterizes the loading time of the largest visible content element on the page; these indicators are considered key predictors of commercial success, directly influencing conversion and the rate of refusals. The third group is related to the measurement of technical debt and is based on indirect indicators such as the dynamics of defects and the duration of the full development cycle, including efforts for code correction and rework.

Results and Discussion

Effective migration of legacy systems presupposes prior recognition that technical debt (TD) does not constitute a homogeneous entity, but decomposes into an aggregate of types, each of which requires differentiated remediation strategies [8]. Under conditions of large-scale system transformation, architectural debt becomes the primary focus of attention, arising at the moment when the basic structure of an application ceases to correspond to the current scale and increasingly complex business requirements. It is precisely architectural constraints, manifested in monolithicity and dependence on obsolete components, that form the greatest resistance to the introduction of modern CI/CD and DevOps practices, hindering both delivery automation and an increase in release frequency [10].

Historically, corporate practice has often been based on trading code quality for accelerated time-to-market of functionality, which has led to the systematic accumulation of TD and the formation of hidden costs that manifest themselves only at the stages of system maintenance and scaling [3]. Quantitative analysis of 39 proprietary production codebases demonstrates that such a compromise is strategically unsound, since deterioration of code quality leads to an increase in operational risks and total cost of ownership. Table 1 presents key indicators illustrating the business effects of low code quality and confirming the instability of a model based on the systematic disregard of technical debt.

Table 1. Comparative analysis of the impact of low code quality on key development indicators and TTM (compiled by the author based on [3]).

Quality/effect metric	Low code quality (Legacy)	High code quality (Migration target)	Quantitative difference
Defect frequency (Defect Rate)	High	Low	15 times more defects
Time to fix issues (Development Speed)	Long (High TTM)	Fast (Low TTM)	124% longer in development
Maximum resolution cycle time (Uncertainty)	High	Low	9 times longer
Loss of developer working time due to TD	Significant	Low	Up to 42% spent on rework

The data presented in Table 1 show that low code quality is transformed into direct organizational losses: up to 42% of developers' working time is spent on fixing defects and refining existing solutions, whereas the time costs for implementing changes increase manifold. Such dynamics has a directly destabilizing effect on the TTM indicator and thereby limits the company's ability to respond promptly to changing market requirements and to strengthen product differentiation [23]. Under these conditions, remediation of technical debt should be interpreted not as an auxiliary engineering activity but as a form of capital investment in the growth of labor productivity and the reduction of aggregate operational risk.

From the standpoint of financial management, technical debt is not subject to complete elimination, since a certain level of it represents a rational price for the option of speed and flexibility of innovation [5]. Instead of attempting the total elimination of TD, the concept of controlled debt is formed: maintaining the digital core in a stable state and ensuring its continuous reinvention presuppose the systematic allocation of about 15% of the total IT budget to structured remediation based on software-autonomous methods of analysis and reduction of debt [4]. Empirical observations show that organizations that consistently adhere to such a balance between the development of new functionality and the planned reduction of TD demonstrate higher revenue growth rates compared to competitors who ignore these practices.

In the architectural dimension, migration of large-scale legacy code represents an operation with an elevated level of risk, especially with regard to revenue-critical systems such as e-commerce platforms. A big-bang transition strategy under such conditions is associated with a high probability of catastrophic failures and prolonged downtime, which makes it practically unacceptable from the point of view of business sustainability.16 An alternative is incremental migration based on the Strangler Fig Pattern, considered as a strategic mechanism for reducing architectural and financial risk.

The pattern proposed by Martin Fowler defines an operational scheme for the stepwise displacement of the monolith [19]. In its software interpretation, SFP presupposes the introduction of an intermediate facade (proxy) between clients and the legacy system. At the initial stage, the facade routes the main bulk of requests to the monolithic application, while new functionality is implemented in a modern technology stack in parallel. As the volume of transferred capabilities grows, the proxy begins gradually to redirect the corresponding requests to the new services until the obsolete system loses critical functions and can be safely decommissioned [15]. This approach makes it possible to distribute risks and capital expenditures over time, maintaining the continuity of business processes and ensuring controlled reduction of technical and architectural debt.

Strangler Fig Pattern (SFP) acts as the central mechanism of risk management and business continuity maintenance during migration. Its use creates the possibility of starting transformation with the least risky or, conversely, the most business-critical components for which strict requirements for scalability or change frequency have already been formulated. Such stepwise capture and replacement of functionality increases team confidence, ensures manageability of changes, and makes it possible consistently to reduce technical debt at the level of individual components rather than the system as a whole. This strategy is organically combined with modern DevOps practices, since it allows parallel coexistence of legacy and new code without disrupting CI/CD pipelines and while maintaining the possibility of regular and predictable releases [10].

Migration to a modern technology stack including ReactJS and TypeScript in this logic is interpreted not only as a way to eliminate historically accumulated architectural debt associated with monolithicity and obsolete dependencies but also as a tool for systemic prevention of the accumulation of new code debt. ReactJS forms a modular, component-based architecture that is initially better adapted to changing requirements and the evolution of the domain than traditional approaches, which strengthens the long-term sustainability of the project [20]. The introduction of TypeScript acquires key importance for technical debt management. As a strictly typed superset of JavaScript, TypeScript performs the function of a disciplining quality control mechanism. Under the conditions of high entropy typical for startups and dynamic e-commerce platforms with poorly defined and constantly changing requirements, the thesis is often voiced that strict typing complicates the initial stages of development [21]. However, the fundamental advantage of static typing is manifested in the phases of refactoring and long-term maintenance: the need for explicit specification of data structures and abstractions radically simplifies the safe introduction of changes, reduces the probability of unintentional formation of new technical debt arising from incomplete understanding of the code, fragmentary documentation, or lack of resources for analysis of the existing implementation [23].

The use of TypeScript acts as a fundamental mechanism for protecting against structural degradation of code, a risk inherent not only to legacy but also to newly created systems, especially under conditions of widespread use of large-scale language models (LLMs) for the generation of software artefacts [25]. The requirement of explicit description of interfaces and contracts in TypeScript minimizes the probability of hidden gaps in logic, reduces the risk of formation of Implementation Laziness, situations in which complex business logic is partially omitted or simplified, including for the sake of fitting into token limits, and thereby preserves the cleanliness of architectural layers. In this context, the choice of the ReactJS/TypeScript combination as the target stack for frontend migration acquires the nature of a strategic decision that ensures not only current modernization but also the long-term sustainability of the system while maintaining a high rate of change.

An in-depth analysis of the migration of long-standing legacy code of the large e-commerce platform Smart Business Technologies (formerly Yandex.Market) to the modern ReactJS/TypeScript stack demonstrates a pronounced relationship between purposeful reduction of technical debt and improvement of critically important operational and business metrics. The migration was carried out incrementally with an emphasis on unification of business logic and transition to a modular component-based architecture. Architectural restructuring combined with elimination of accumulated code and architectural debt significantly reduced Time to Market (TTM). Before the beginning of the transformation, high code complexity, low cohesion of modules, and brittle dependencies made the deployment of new functions a lengthy and hard-to-predict process [10]. The transition to modular architecture and introduction of strict static typing led to refactoring becoming substantially safer and to acceleration of development and deployment cycles, which was expressed in an increase in release frequency. Reduction of TTM in this context is interpreted as a direct indicator of increased organizational flexibility and the ability of teams to respond quickly to feedback and market changes [12].

The most significant quantitative effects were recorded in the area of user experience metrics included in Core Web Vitals. The legacy frontend built on outdated approaches exhibited inefficient rendering and increased delays, which negatively affected both Largest Contentful Paint (LCP) and Time to Response/Interactive (TTR). After migration and targeted optimization of the frontend layer, the values of these indicators were substantially improved. LCP, which measures the loading time of the main content element of the page, serves as a key parameter, since it directly determines the user's subjective perception of system speed and thereby influences conversion and the probability of refusal of further interaction with the platform (see Table 2).

Table 2. Summary metrics of improvement of operational indicators after migration to a modern technology stack (compiled by the author based on [10, 11, 22])

Indicator	Before migration (Legacy code)	After migration (ReactJS/TS)	Improvement (Illustrative data)	Business implication
Time to Market (TTM)	High	Reduced	Significant reduction	Increased innovation speed and competitiveness
Largest Contentful Paint (LCP)	Low	Significantly improved	Substantial acceleration	Increase in conversion rate and reduction in bounce rate
Time to Interactive (TTR)	High latency	Optimized	Reduction of delays	Improved user experience and retention
Reduction of critical errors (per month)	High	Substantially reduced	Significant reduction	Decrease in operational risks and MTTR

The improvement in LCP and TTR achieved as a result of migration represents not so much a local technical success as a direct lever of monetization. In the highly competitive e-commerce segment, interface loading speed is directly translated into revenue, and empirical data from large platforms convincingly confirm this relationship. Thus, Vodafone recorded an 8% increase in sales on an optimized page where the LCP value was 31% better compared to the baseline version; a Propellernet study showed that faster sessions with improved Core Web Vitals are 34% more likely to lead to conversion; the company Agrofy, having optimized LCP and other Core Web Vitals indicators, achieved a 76% reduction in the rate of refusals, from 3.8% to 0.9% on product detail pages; historical data from Amazon demonstrate that every additional 100 ms of page load delay cost 1% of lost sales [22].

The totality of these results shows that investments in the elimination of technical debt affecting frontend performance have a high and quantitatively measurable return on investment (ROI). In this context, improvement of LCP acts as a key translational metric that makes it possible directly to convert engineering efforts in optimization and remediation of technical debt into financially significant business indicators (see Fig. 1).

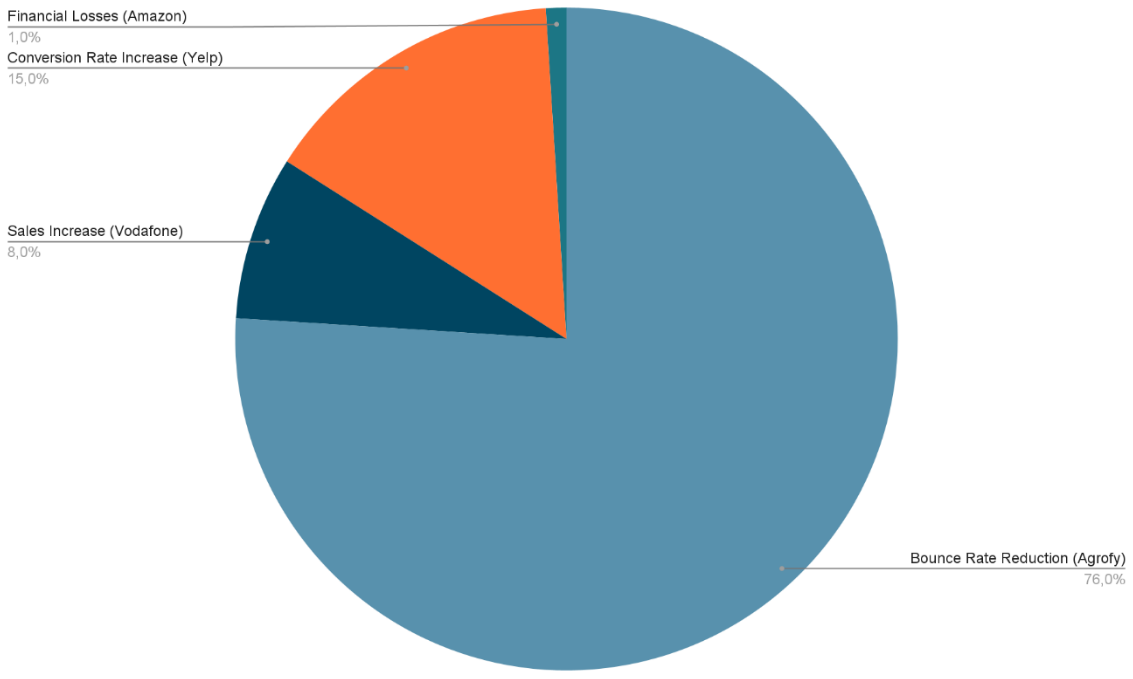

Fig.1. Quantitative business impact of key web metric optimization (LCP/TTI) (compiled by the author based on [22]).

As illustrated in Figure 1, performance optimization is directly reflected in the dynamics of key business indicators: successful migration to a modern technology stack manifests itself not only in the internal effect in the form of a reduction in technical debt (a decrease in the number of errors, acceleration of TTM), but also in the external, user dimension, expressed in the improvement of LCP and an increase in conversion. Verification of empirical data confirms the author hypothesis and makes it possible to develop a holistic model of technical debt management in the course of migration, in which architectural and code aspects are regarded as interrelated elements of a single strategy.

The formulated strategy, based on the incremental Strangler Fig Pattern (SFP) approach, combines two key vectors of working with technical debt: the sequential elimination of accumulated architectural debt and the systemic prevention of the emergence of new code debt. The Strangler Fig pattern provides the possibility of evolutionary migration of large-scale systems without interruption of business-critical operations, translating the process into a predictable iterative mode, which is fundamentally important for high-load e-commerce platforms [15]. In this way, controlled, stepwise reduction of architectural debt is achieved. In parallel, the introduction of a disciplined technology stack, primarily TypeScript, functions as a mechanism of internal hygiene of the code, preventing the accumulation of new code debt in the created application and consolidating the effect of migration over the long term [20]. Figure 2 presents a demonstration of the contribution of strategic decisions to long-term maintainability

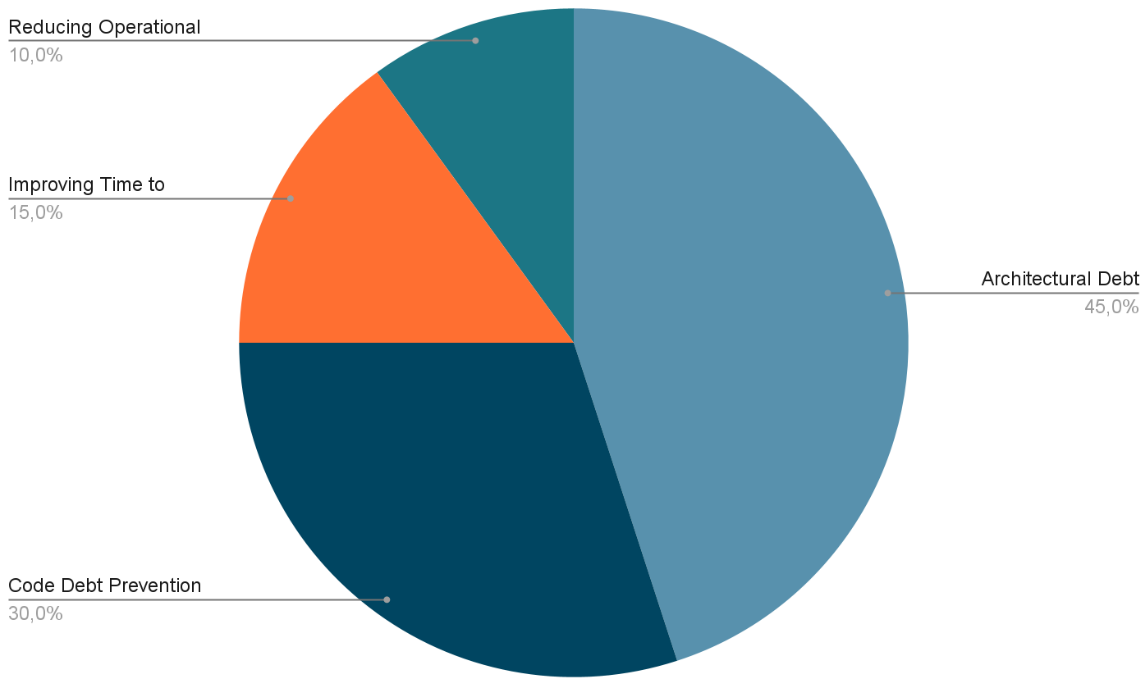

Fig.2. Structural contribution of strategic migration components to overall technical debt reduction (compiled by the author based on [20]).

Figure 2 demonstrates that it is precisely the architectural strategy based on the application of SFP that acts as the main driver in reducing already accumulated technical debt, whereas preventive mechanisms ensured, in particular, through the use of TypeScript form the basis of long-term sustainability and prevent renewed accumulation of debt in the updated system.

The implementation of such a strategy, however, is associated with a number of critical challenges. First, a significant limitation is the organizational barrier: successful migration requires not only high engineering competence but also full-scale support of the initiative at the executive level (executive buy-in), as well as close cross-functional interaction between business and technical units [22]. Misalignment of priorities between short-term business goals and the tasks of deep technical remediation leads to a situation in which even technically sound initiatives may become marginalized and fail to deliver the expected effect [12]. Second, substantial complexity is posed by the parallel operation of two systems, the legacy and the new one, over a prolonged period of incremental migration. Maintaining such a dual infrastructure requires high discipline in DevOps processes, extensive use of automated regression testing to guarantee invariance of the behavior of critical legacy functions, as well as strict data management and consistency between the environments [10]. Third, the risk of accumulation of new debt persists: even when using modern tools such as TypeScript, insufficiently strict architectural control can lead to architectural erosion [25]. When architectural discipline is weakened, developers may bypass strict constraints, create implicit dependencies and local workaround solutions, which accelerates the formation of structural technical debt in new microservices and undermines the effect of migration. This underlines the need to consider technological modernization in conjunction with continuous architectural audit and regular review of design principles.

Taken together, migration based on the SFP pattern and the ReactJS/TypeScript stack demonstrates the transformation of technical debt from an uncontrolled latent obligation into a managed strategic asset. The achieved improvement of TTM indicators and, most notably, LCP and TTR shows how targeted engineering investments are converted into a sustainable competitive advantage and a measurable increase in conversion.

Conclusion

The results of the conducted study demonstrate that technical debt management in the context of migration of long-standing legacy systems must rely on a strategic, financially justified approach that goes far beyond purely technical optimization. The aim formulated in the work, the development of an integrated approach linking architectural strategy, technical debt prevention, and quantitatively measurable business metrics, can be considered achieved, since the proposed model is confirmed by both theoretical analysis and empirical data.

The key theoretical and practical conclusion is that the incremental architectural Strangler Fig Pattern is the most rational migration strategy for large-scale systems, as it makes it possible to minimize the aggregate operational and financial risks of transition. In combination with the introduction of a modern disciplined technology stack such as the ReactJS/TypeScript combination, this approach forms a two-sided contour of debt management: on the one hand, it ensures systematic elimination of accumulated architectural debt; on the other, it prevents the emergence of new code debt, simplifying refactoring and maintenance under conditions of high volatility of requirements.

Empirical analysis of the migration case of a large e-commerce platform confirmed that such a strategy leads to statistically significant and business-meaningful results: a reduction in Time to Market and critical improvement of user experience quality metrics (LCP and TTR) is recorded. Improvement of LCP in this context manifests itself as a convertible financial effect, a decrease in the rate of refusals, an increase in conversion and, as a consequence, an increase in the profitability of the digital channel. In this way, engineering decisions and efforts for technical debt remediation receive direct expression in metrics that are relevant for managerial and investment decisions.

The hypothesis that strategic incremental migration using SFP and static typing (TypeScript) leads to a statistically significant improvement in TTM and LCP/TTR indicators is confirmed on the basis of the collected data. In this framework, technical debt ceases to be solely a passive burden and is transformed into a managed strategic asset, the quality of management of which directly determines the competitiveness of the organization.

Promising directions for further research are primarily related to the development and validation of automated tools for continuous monitoring of architectural erosion in modern microservice architectures built using ReactJS/TypeScript, as well as to the refinement and formalization of methods for quantitative measurement of TTM. Of particular relevance is the creation of more universal and operationalized metrics that reflect the impact of technical debt on the duration of the full development and deployment cycle in environments with actively used CI/CD pipelines.

References

1. eLuminous Technologies. (2025, October 10). The strategic guide to technical debt 2025. eLuminous Technologies. https://eluminoustechnologies.com/blog/technical-debt/ (date accessed: November 05, 2025)

2. Zartis. (2025). Why 2025 is the year tech debt becomes a strategic risk. Zartis. https://www.zartis.com/why-2025-is-the-year-tech-debt-becomes-a-strategic-risk/ (date accessed: December 01, 2025)

3. Tornhill, A., & Borg, M. (2022, May). Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In Proceedings of the International Conference on Technical Debt (pp. 11-20). https://doi.org/10.48550/arXiv.2203.04374

4. Accenture. (2024). What is tech debt? Accenture. https://www.accenture.com/us-en/insights/what-is-tech-debt (date accessed: October 12, 2025)

5. Accenture. (2024). Build your tech and balance your debt. Accenture. https://www.accenture.com/content/dam/accenture/final/accenture-com/document-3/Accenture-Build-Your-Tech-and-Manage-Your-Debt-2024.pdf (date accessed: November 15, 2025)

6. Fürnweger, A., Auer, M., & Biffl, S. (2016). Software evolution of legacy systems: A case study of soft-migration. In Proceedings of the 18th International Conference on Enterprise Information Systems (ICEIS 2016) (pp. 413–424). https://doi.org/10.5220/0005771104130424.

7. Fairbanks, G. (2020). The pragmatic designer: Ur-technical debt. IEEE Software, 37(4). https://doi.org/10.1109/MS.2020.2986613.

8. Kong. (2025, August 5). The roadmap for reducing technical debt in 2025. Kong Blog – Learning Center. https://konghq.com/blog/learning-center/reducing-technical-debt (date accessed: October 25, 2025)

9. Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101, 193–220. https://doi.org/10.1016/j.jss.2014.12.027.

10. Technical debt and DevOps: Strategies for managing legacy systems in a CI/CD world. (2023). NeuroQuantology, 21(7). https://doi.org/10.48047/nq.2023.21.7.nq23115.

11. Product Lab / Rocketech. (n.d.). Time to Market: руководство для стартапа по сокращению времени выхода на рынок. Product Lab. https://productlab.ru/blog/time-to-market (date accessed: October 07, 2025)

12. MachineHeads. (2025, April 16). Time to Market: что это значит и как с этим работать. MachineHeads. https://machineheads.ru/blog/metrika-time-to-market-derzhite-vremya-pod-kontrolem/ (date accessed: November 10, 2025)

13. TechDebt 2025. (2025). TechDebt 2025 – International Conference on Technical Debt. conf.researchr.org. https://conf.researchr.org/home/TechDebt-2025 (date accessed: October 30, 2025)

14. TechDebt Steering Committee. (2025). International Conference on Technical Debt conference series – TechDebt. conf.researchr.org. https://conf.researchr.org/series/TechDebt (date accessed: November 08, 2025)

15. Microsoft. (2025, February 19). Strangler Fig pattern – Azure Architecture Center. Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig (date accessed: October 22, 2025)

16. Buxton Consulting. (2024). Reducing technical debt during system migrations: A strategic blueprint. Buxton Consulting. https://buxtonconsulting.com/general/reducing-technical-debt-during-system-migrations-a-strategic-blueprint/ (date accessed: November 25, 2025)

17. Kirkila, L., Klotina, M., Sproge, J., & Romanovs, A. (2025). Case study review: IT legacy system migration success factors. Conference paper. https://doi.org/10.1109/ITMS67030.2025.11236637.

18. Sklavenitis, D., & Kalles, D. (2025). A scoping review and assessment framework for technical debt in the development and operation of AI/ML competition platforms. Applied Sciences, 15(13), 7165. https://doi.org/10.3390/app15137165.

19. Amazon Web Services. (n.d.). The strangler fig pattern – AWS Prescriptive Guidance. AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-aspnet-web-services/fig-pattern.html (date accessed: October 14, 2025)

20. Rafalski, K. (2025, June 30). TypeScript vs React: Which technology is right for you? Netguru Blog. https://www.netguru.com/blog/typescript-vs-react (date accessed: October 28, 2025)

21. u/OP (Reddit username). (2024). Typescript seems to generate a lot of technical debt.. Am I doing it wrong? Reddit – r/typescript. https://www.reddit.com/r/typescript/comments/1gzpclv/typescript_seems_to_generate_a_lot_of_technical/(date accessed: November 19, 2025)

22. Conductor. (n.d.). Page speed matters: 10 case studies show why. Conductor Academy. https://www.conductor.com/academy/page-speed-resources/ (date accessed: October 05, 2025)

23. Paddle. (2024, July 31). Impact of technical debt: How to identify and reduce it. Paddle Resources. https://www.paddle.com/resources/technical-debt (date accessed: November 14, 2025)

24. Strangler fig pattern. (n.d.). In Wikipedia. https://en.wikipedia.org/wiki/Strangler_fig_pattern (date accessed: October 27, 2025)

25. Slater, T. (2025, December 3). Quantitative analysis of technical debt and pattern violation in large language model architectures. https://doi.org/10.48550/arXiv.2512.04273

26. Polcode. (2025, January 15). Legacy modernization strategies and approaches for 2025. Polcode Blog. https://polcode.com/resources/blog/legacy-modernization-strategies-approaches/ (date accessed: October 17, 2025)

27. Bishop, V. III, & Simske, S. J. (2024). Evaluating software contribution quality: Time-to-modification theory. Preprint. https://doi.org/10.48550/arXiv.2410.11768

28. Bishop, V. III, & Simske, S. J. (2024, October 15). Evaluating software contribution quality: Time-to-modification theory. arXiv. https://doi.org/10.48550/arXiv.2410.11768.

29. Deloitte. (2023). Life after debt: Venture debt funding could grow again in 2024. Deloitte Insights. https://www.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions/2024/technology-venture-debt-prediction.html (date accessed: October 16, 2025)

30. Deloitte. (2023). 2024 technology industry outlook. Deloitte Insights. https://www.deloitte.com/us/en/industries/tmt/articles/technology-industry-outlook.html (date accessed: November 02, 2025)

31. Saritasa. (2025, August 25). Legacy software modernization in 2025: Survey of 500+ U.S. IT pros. Saritasa Insights. https://www.saritasa.com/insights/legacy-software-modernization-in-2025-survey-of-500-u-s-it-pros (date accessed: November 18, 2025)