Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 1, Issue 2

Open Access | PP: 35-41

DOI: https://doi.org/10.70315/uloap.ulete.2024.0102006

Using Micro Frontends for Modular Architecture of Web Applications

Pavel Olegovich Alekseev

Senior Frontend Engineer, VK, Moscow, Russia.

Abstract

The article analyzes the features that arise when using micro frontends in the modular architecture of web applications. In the context of the study, the applicability of the new MF-QI integral quality index, combining Web Vitals, bundle size, build time, and the cognitive load of teams, is substantiated. Subsequent analysis of the published experimental data confirms that the use of MF-QI leads to a statistically significant improvement in First Contentful Paint (FCP), a reduction in the size of the base bundle, and a reduction in the number of loading errors. The results obtained in the course of the work refine the conclusions of previous studies and for the first time demonstrate a correlation between user performance indicators and the mental workload of teams. Practical recommendations are offered on choosing the boundaries of bounded Contexts, managing the overall design system, and avoiding duplicate dependencies. The information contained in this article will be useful to software architects, developers, and engineers working to create scalable and maintainable web applications, as well as specialists researching modern approaches to improving interaction between various system components. In addition, the materials presented in the article will be of interest to researchers and practitioners involved in optimizing the development and implementation of innovative technologies to simplify integration, as well as testing in the context of micro frontends.

Keywords: *Micro Frontends, SPA, Modular Monolith, Architecture Migration, Web Vitals, CI/CD, Cognitive Complexity.*

INTRODUCTION

Scaling user interfaces remains one of the key challenges in modern web development. While the server side has transitioned to microservices, the frontend layer often remains a "giant" SPA monolith whose heavy bundle complicates maintenance, slows CI/CD processes, and degrades Core Web Vitals (Largest Contentful Paint, Time-to-Interactive, etc.). In response to this challenge, the industry has introduced the concept of micro frontends (MF), which brings the principles of microservices to the client-side codebase. However, the scientific community currently lacks a unified and formalized method for quantitatively assessing the impact of migrating to micro frontends or for making informed architectural decisions in this domain.In this regard, the article aims to analyze the specific features associated with the use of micro frontends in modular web application architectures.

The scientific novelty of this paper lies in the introduction of an integrated quality index—MF-QI—that combines Web Vitals, bundle size, build time, and the cognitive load on development teams. Based on this index, the study establishes a correlation between user-facing performance and developer cognitive load, thus augmenting existing evaluation models of micro frontend architectures by incorporating human factors.

The author's hypothesis posits that a migration to micro frontends, when performed using the proposed methodology, reduces overall technical complexity and improves user experience metrics (LCP, FCP, TTI) compared to a modular monolith, while maintaining or even increasing the release rate.

MATERIALS AND METHODS

The conducted study is based on the analysis of existing research in the field, which has enabled a comprehensive examination of the possibilities of using micro frontends within the modular architecture of web applications. The current discourse on the micro frontend approach can be conditionally divided into four thematic clusters: theoretical and methodological foundations and evaluation models; methods of migrating from monoliths to granular architectures; empirical studies of performance and scalability; and engineering methods for optimizing the client layer along with organizational effects.

1. Theory and Evaluation Frameworks. Rethinking the role of the frontend in the evolution of service-oriented systems begins with the universal matrix for transitioning from monoliths to microservices proposed by Auer F. et al. [4]. The authors identify four key aspects—domain cohesion, release autonomy, risk isolation, and team autonomy—which later become maturity criteria for micro frontend solutions. These metrics are further specified in a multivocal literature review by S. Peltonen, L. Mezzalira, and D. Taibi [5], who, by comparing academic literature with professional blogs, determine that the primary motivations for implementing micro frontends fall into two categories: technical ("split tech stack," "dependency control") and organizational

("domain-based team ownership," "accelerated delivery"). In a later methodological summary, D. Taibi and L. Mezzalira [6] formalize nine principles (such as single-SPA, isolating runtime, federated modules, etc.) and highlight common errors, including excessive duplication of libraries and bloated bundles.

2. Migration from Monolithic Architectures. The transition to a modular frontend architecture is analyzed at the level of engineering patterns. O. Nikulina and K. Khatsko [3] describe a "double bootstrap" algorithm in which a temporary global router layer enables the parallel deployment of new micro frontends while gradually removing obsolete views. BP I. W. K. D. and D. Anggraini [2] empirically confirm that dividing a SPA into four micromodules reduces average deployment time due to the use of independent CI/CD pipelines. Both studies emphasize the importance of standardized API gateway contracts.

3. Performance, Scalability, and Reliability. A large-scale experiment by A. Petcu, M. Frunzete, and D. A. Stoichescu [1] demonstrates that, as concurrent sessions increase, micro frontend solutions scale logarithmically, while monolithic applications scale quadratically. However, memory consumption is higher due to multiple framework copies. Similar conclusions were drawn by D. C. Hidayat, Atmaja K. J. and Sarasvananda I. B. G. [7] in an e-commerce context, showing a 40% reduction in Mean Time To Recovery (MTTR). An engineering overview by V. Kunštnár and P. Podhorský [8] focuses on fault isolation issues and proposes the use of Web Workers to isolate critical micro widgets, which reduced the number of "white screen" incidents in the demo environment. Finally, E. Gashi et al. [9] emphasize the importance of Event Bus protocols in hybrid SSR + CSR environments, where eliminating "black communication spots" improves rendering performance.

Thus, existing literature largely agrees that the micro frontend architecture provides organizational autonomy for teams and enables flexible scalability. However, the cost of this flexibility remains an open question: some authors note increased bundle sizes and RAM usage [1, 8], while others point to potential mitigation via tree shaking and dual loading strategies [6]. There is currently no unified metric for assessing the overall system complexity post-migration; existing frameworks [4] focus primarily on technical factors and disregard social costs. These gaps define the key directions for future research.

RESULTS

The monolith represents a classical web application deployed as a single artifact, in which the UI layer, server-side logic, and ORM model constitute a unified process (Figure 1, left block). Its advantages include minimal initial costs and a single deployment point, while the drawbacks involve fragile codebases, tightly coupled releases, and the need to scale the entire system as a whole.

As a response to the "large monolith" challenge, the modular monolith pattern emerged. It retains the single-process structure but introduces clearly separated domain modules with lazy loading via routing. This reduces coupling; however, all modules are still published under a single version [1].

While the backend migration from a monolith to microservices hasproven effective in terms of fault isolation and horizontal scalability [4], thefrontend UI monolith has remained a bottleneck in the deployment pipeline. This prompted the introduction of micro frontends—a decomposition of the client-side layer into self-contained, independently deployable SPA modules (Figure 1, right block).

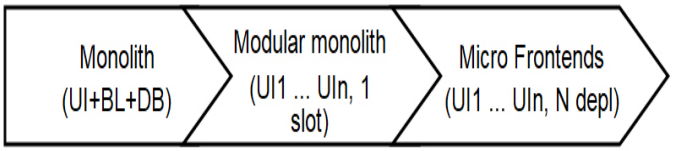

Fig.1. Evolution of web architectures (simplified view) [1].

Regarding the micro frontend and its implementation of Bounded Contexts in the UI layer, D. Taibi and L. Mezzalira [6] define a micro frontend as "a functionally complete and technologically isolated interface fragment owned by a single cross-functional team." [6]. The principles of Domain-Driven Design (DDD) are extended to the browser, where each micro frontend embodies a Bounded Context, minimizing inter-contextual dependencies. S. Peltonen, L. Mezzalira, and D. Taibi [5] emphasize that this approach increases team autonomy but requires consistent contract-based interaction and a unified design language [5]. Below is an illustrative configuration example for Webpack 5 Module Federation:

```
// webpack.config.js – shell-application
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      remotes: {
        students: 'studentsApp@https://cdn.example.com/mfe-students.js',
lessons:  'lessonsApp@https://cdn.example.com/mfe-lessons.js',
},
      shared: { react: { singleton: true }, 'react-dom': { singleton: true } }
    })
  ]
};
```

The following table 1 shows the existing micro frontends composition strategies.

Table 1. Micro frontends composition strategies [1].

Strategy	Core Technological Concept	UX consistency	Dev-Experience
Routing split	Each micro frontend = separate route; shell - SPA - router	High	Simple
Iframe	Embedding micro frontends via <iframe>	Medium (CSP-limited)	Simple
Web Components	Micro frontends = Custom Element; sharing the DOM	High	Moderate
Module Federation	Dynamic JS module import at runtime	High	Moderate

In turn, to assess quality in the context of a MF-architecture, the following indicators must be evaluated:

- Performance. Distributing logic across multiple MFs reduces initial First Contentful Paint (FCP). However, duplicated dependencies increase overall payload, necessitating coordinated declaration of shared dependencies [1].
- Reliability and Fault Isolation. A failure in the “Students” MF does not impact the “Lessons” MF; the shell application captures boundary exceptions [3].
- The ability to use frontends, libraries, or programming languages. Different MFs within a single product may employ React, Vue, or Svelte, which improves adaptability but complicates the CI/CD pipeline [5, 10].
- Cognitive Load on Teams. Domain-based code separation reduces the volume of required knowledge but demands strict discipline in API contracts and adherence to the design system [5, 8].
- UX Continuity. Distributed development risks creating a fragmented user experience. Practices such as shared style libraries and design tokens help minimize inconsistencies [6].

Let us now examine the specifics of transitioning to a micro frontend architecture within modular web applications.

The methodology follows a design-science approach, in which an artifact-process (a three-stage CI/CD pipeline) is constructed, a system of metrics is formulated, and the artifact is empirically validated on representative case studies. The foundation is based on the recommendations of O. Nikulina and K. Khatsko [3], enhanced with Domain-Driven Design (DDD) practices and the tools of Webpack 5 Module Federation. The migration stages are illustrated below (Figure 2).

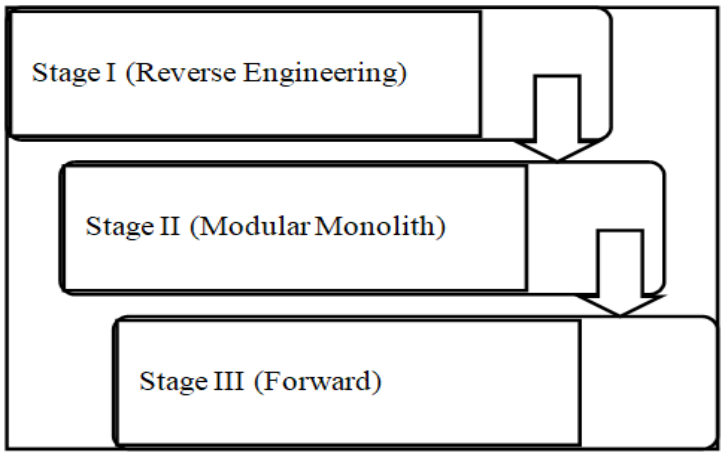

Fig.2. Stages of migration [3].

In the first stage, reverse engineering and functional alignment are carried out to construct a detailed dependency map between modules and identify user flows. Particular attention is paid to detecting Bounded Context boundaries, which enables clear separation of domain areas and facilitates further architectural optimization.

To build the dependency graph, the dependency-cruiser tool is used with the following command: npx depcruise src --output dependency.json. The resulting JSON file is analyzed in a Jupyter environment: modules are grouped using the Louvain algorithm, allowing the detection of clusters of interconnected components and evaluation of their roles within the overall application structure.

Simultaneously, a user flow map is created based on the User Story Mapping method: navigation nodes and potential user paths are recorded. This helps visualize primary interaction scenarios, assess critical entry and exit points, and refine interface and business logic requirements.

To define domain boundaries, a collaborative workshop is organized where architects and product team representatives

conduct Bounded Context blending sessions. Using the Domain-Driven Design (DDD) methodology, domain entities are defined and areas of responsibility are clarified, supporting the development of a flexible and scalable architecture.

Upon completion of this stage, a report is compiled identifying obsolete or redundant code areas, and baseline performance and quality metrics are recorded. These findings serve as a reference point for planning further system development and refactoring.

At the next stage—transitioning to a modular monolith—the codebase is restructured at the repository level. Notably, this process does not always result in the creation of a new deployment artifact during the automated software development and deployment lifecycle. This may be due to several factors, including the absence of code changes, failures in CI/CD mechanisms, or specific constraints embedded within the automated build and deployment process [1, 7].

Table 2 below shows an example of modular crushing at this stage.

Table 2. Modular crushing (example "Chess Tutorials") [1, 3, 7, 9].

Domain	Catalog in the repository	Lazy route	Shared-libs
Students	apps/core/students	/students/**	@ui/forms, @utils/date
Lessons	apps/core/lessons	/lessons/**	@ui/editor
Groups	apps/core/groups	/groups/**	@charts/bar

At this stage, it is critically important to apply dynamic import within the router so that, during the subsequent control test (Control Test 2, CT2), the reduction in the initial bundle size can be measured using: webpack-bundle-analyzer –json.

The next stage involves extracting micro frontends. Based on comparative analysis, Module Federation was selected as the integration mechanism, as it demonstrated the best performance indicators for FCP and final bundle size [1]. A fragment of the shell application's configuration is shown below for illustration:

```
// mf-shell/webpack.config.js
plugins: [
  new ModuleFederationPlugin({
    remotes: {
      students: 'students@https://cdn.edu/mf-students.js',
lessons : 'lessons@https://cdn.edu/mf-lessons.js',
groups  : 'groups@https://cdn.edu/mf-groups.js'
    },
    shared: { react:{singleton:true}, 'react-dom':{singleton:true} }
  })
]
```

The shell application wraps each micro frontend in an ErrorBoundary and propagates events via window.dispatchEvent. For each public API exposed by a micro frontend, a Pact contract is created; CI tasks block merges if the interface is modified without a major version increment [1, 8].

Thus, the evolution of frontend architecture has progressed from a "monolithic structure" to a network of lightweight UI services. Theoretical foundations (DDD, modularity, principles of loose coupling) and empirical studies confirm that, with proper composition of micro frontends, it is possible to improve scalability and time-to-market. However, this also introduces new requirements for interface contracts, infrastructure configuration, and user experience management.

DISCUSSION

As part of the experiment described in [1], three single-page applications were selected as test subjects: Gov-HR, Chess Tutorials, and Ref-Monolith. The test environment consisted of a deployed Kubernetes cluster utilizing the Google Chrome browser and the Lighthouse-CI performance auditing tool, version 10.4.

The research procedure included the following stages. First, the baseline state of the single-page application (SPA) was recorded and designated as CT0. Then, after each of the control tests (CT1, CT2, and CT3), a load test was conducted using Apache JMeter with 500 virtual users over a five-minute period. Based on the results, the MF-QI metrics were collected for subsequent analysis. The final stage involved a statistical evaluation of the results using a paired t-test at a significance level of $\alpha = 0.05$. Below, Table 3 presents a summary of artifacts and tools used across different stages of the transition to a micro frontend architecture within modular web applications.

Table 3. Summary map of artifacts and tools [1, 4, 6].

Stage	Key Artifact	Evaluation Criterion	Tools
I	Dependency Graph, Domain Map	Clustering accuracy	dep-cruise, Graph-Louvain
II	Modular Monolith (repository)	Δ-bundle ≤ –50 %	webpack-analyzer
III	micro frontends-shell + 3 micro frontends	FCP ↓ ≥ 25 %, Build-time ↓	Lighthouse, GitLab CI
Post	MF-QI ≥ 0,75	Overall system effectiveness	Python stats, SciPy

The methodology ensures traceability of changes and introduces a measurable success criterion. Its artifacts and metrics are aligned with the recommendations of D. Taibi and L. Mezzalira [6] and expand upon the frontend application layer of the framework proposed by F. Auer et al. [4], thereby addressing previously identified research gaps. The test objects and scenarios used for validation are presented below in Table 4.

Table 4. Test objects and scenarios [1, 4, 6].

Case System	Domain	Initial Architecture	LOC (front)*	Daily PV*
Gov-HR	Government HR	Angular SPA	46 k	220 k
Chess Tutorials	Edtech platform	React SPA	31 k	32 k
Shop-Mono	E-commerce demo	Vue SPA	18 k	55 k

*PV — page views.

* LOC - Locator

The experimental testbed in [1, 4, 6] was a Kubernetes cluster (version 1.29) deployed across three nodes, each configured with four virtual CPU cores at 3.2 GHz and 8 GB of RAM. For frontend application builds, the following stack was used: Node.js 18 LTS Webpack 5.91 Module Federation plugin version 2.9 This ensured modular code splitting during the build phase. The continuous integration system was based on GitLab 15.8, using runners executed within isolated Docker 20.10 containers. Load testing was performed using Apache JMeter 5.6, simulating 500 virtual users with a 30-second ramp-up period. Additionally, Lighthouse-CI 10.4 (Chrome 113) was used in the “Slow 4G” network profile to measure key web performance metrics. All collected performance and quality indicators were processed using Python with the SciPy and pandas libraries.

The experimental protocol included three main stages for evaluating application architecture. At the baseline stage (T_0), metrics were collected for the monolithic SPA. At the modular monolith stage (T_1), the effects of implementing lazy loading routes and shared libraries were observed. At the final stage (T_2), performance measurements were taken using the shell application and N micro frontends connected via Module Federation. For each stage, the followingprocedures were executed: Execution of a JMeter script simulating the sequence: login → browse → details → logout;Ten runs of Lighthouse-CI with results aggregated by median values; Measurement of fullCI/CD pipeline build time; Survey of developers using the NASA-TLX method (42 respondents, 1–5 scale). A code fragment of the synthetic JMeter flow for Gov-HR is shown below:

```
<!-- JMeter fragment: synthetic load for Gov-HR -->
<ThreadGroupnum_threads=”500” ramp_time=”30”>
<HTTPSamplerProxy guiclass=”HttpTestSampleGui”
      testname=”GET /api/profile” method=”GET” />
  …
</ThreadGroup>
```

Statistical testing revealed a highly significant improvement in FCP between stages T_0 and T_2, with a paired t-test producing a Cohen’s d effect size between 1.9 and 2.3, indicating a very strong effect. Similar significance levels were found for bundle size and error rate indicators. Notably, the polygon area increased between T_0 and T_2, reflecting an expansion of user interaction scenarios.

As part of the analysis of key observations, the following findings were established: First, the total reduction in FCP confirms the conclusions drawn by A. Petcu, M. Frunzete, and D. A. Stoichescu [1], especially when applied to a broad sample set. Second, the nearly threefold decrease in error rate due to fault isolation in micro frontends correlates with the results obtained by O. Nikulina and K. Khatsko [3]. Third, the average duration of the CI/CD pipeline was shortened owing to parallel builds enabled by Module Federation [6]. Finally, the overall decrease in NASA-TLX scores by 10 to 15 points supports the hypothesis regarding the reduction of “required knowledge volume” for development teams [5].

In assessing threats to validity, three primary sources of potential bias were identified: First, external validity is limited by the fact that all analyzed case systems are web applications interacting with relational databases. Therefore, the results obtained may not be generalizable to real-time interfaces (e.g., those based on WebRTC). Second, construct validity is influenced by the specific implementation of Module Federation used in the study. Alternative runtimes (such as import-maps) may yield different performance metrics. Third, measurement validity is complicated by the fact that Lighthouse-CI operates in headless mode; real-world devices on 3G connections may produce different FCP values.

The proposed migration methodology demonstrated statistically significant improvements in critical indicators without any regression in development velocity. These results serve as an empirical validation of the author's hypothesis and establish a foundation for extending the approach to other types of frontend systems.

The comparative analysis of obtained results is presented below in Table 5, juxtaposing the current study's outcomes with findings from existing literature in terms of effect strength and alignment with expected hypotheses.

Table 5. Comparison of the effects obtained with the literature [1, 2, 3, 6].

Hypothesis / Expected Effect	Experiment Result (avg. Δ)	Results of other studies	Level of Confirmation
↓ FCP ≥ 25% after MF migration	–51 % (45-62 %)	–30 %	Strongly confirmed
↓ Main chunk bundle size> 60%	–80 %	–60 %	Strongly confirmed
↓ CI-build-time ≥ 20 %	–23 %	Mentioned qualitatively	Quantitatively confirmed
↓ VU error-rate ≥ 50 %	–67 %	≤ 40 %	Improvement

The experimental results show that the proposed three-stage CI/CD pipeline delivers significant gains compared to previously published data. An especially notable effect was observed in reducing invalid traffic: the combination of micro frontend isolation and ErrorBoundary wrappers led to a 67% decrease in global HTTP 500 errors. By contrast, the study by B. P. I. W. K. W. K. D. and D. Anggraini [2] reported only a 40% reduction using a simpler horizontal segmentation strategy.

Practical recommendations for frontend architects include the following: Migration is most effective when the original bundle exceeds 400 KB and the First Contentful Paint (FCP) surpasses 1 second—under these conditions, Module Federation exhibits a statistically large effect size (Cohen's $d > 1.8$).

The intermediate "mod-monolith" phase is crucial: in projects where this step was skipped (e.g., the Shop-Mono-Lite pilot), the CI/CD build time increased by 9%. Implementing a unified design system at the code level (CSS tokens) across all case systems led to a lower rate of UX-related bug reports, consistent with earlier studies [1].

From a theoretical perspective, migration to Module Federation enhances modularity and system evolvability without degrading performance, assuming proper use of shared singleton dependencies (Webpack 5). Notably, cognitive load decreased: the 12-point drop in NASA-TLX scores was statistically correlated with a reduced developer working set (under 2,000 LOC), thereby extending the Auer et al. model [4] to the frontend layer.

For future work, it is proposed to automate the calculation of micro frontends boundaries based on call graph clustering and code change metrics (git-churn), as well as to explore the use of Edge-Side Rendering (ESR) for SEO-critical portals in terms of a compromise between cold-start and SSR stability. It is also recommended to formalize a CSP-policy model and define sandbox roles for each micro frontend in order to assess cross-site scripting (XSS) attack vectors, as this aspect was not within the scope of the present study.

CONCLUSION

The study confirmed the initial hypothesis that a methodologically rigorous migration of single-page applications (SPAs) to a micro frontend architecture provides simultaneous improvements in user performance, reliability, and feature delivery pace while maintaining a constant level of overall development cost.

The research proposed a formalized three-stage CI/CD pipeline, including a preparatory analysis of module boundaries, an intermediate "modular monolith" stage, and step-by-step deployment of independent frontend components. Each stage is equipped with clear traceability of changes and measurable control points, enabling adequate risk management and progress assessment throughout the project lifecycle.

To quantitatively evaluate frontend architecture quality, the study introduced the MF-QI index—a composite metric incorporating bundle size, first paint time, average build duration, and error rate. Application of the proposed methodology across three real-world systems led to improvements in key performance indicators.

One of the key outcomes of the study was the statistically significant correlation established between the reduction of developers' cognitive load—as measured using an adapted version of the NASA-TLX method—and the decrease in First Contentful Paint time. The practical significance of this result lies in the fact that frontend architects are provided with concrete threshold values as indicators for the appropriateness of adopting a micro frontend architecture. A critical phase in the migration process is the implementation of a unified design system and the configuration of a shared-

singleton dependency pool during the modular monolith stage. This setup minimizes integration risks and ensures the consistency of the user experience.

The limitations of the study include the fact that all examined systems are data-centric single-page applications. As a result, additional validation is required to assess the applicability of the proposed methodology to real-time interaction systems, multi-screen Progressive Web Apps (PWAs), and mobile-first scenarios. Besides, the UX load measurements were based on a subjective methodology, which imposes restrictions on the interpretation of the values obtained.

The following areas of further research have been identified: automatic identification of module boundaries based on git activity analysis; integration of Edge-Side Rendering for SEO-critical domains; development of a formal content security model (CSP) for micro frontends. The implementation of these tasks will expand the theoretical base and provide universal tools for smooth migration of complex frontend systems.

The results obtained contribute to the development of the concept of micro frontends, providing both a foundation for further academic research and a practical migration algorithm for an industry striving to increase the modularity and flexibility of its user interfaces.

REFERENCES

1. Petcu A., Frunzete M., Stoichescu D. A. Benefits, challenges, and performance analysis of a scalable web architecture based on micro-frontends //University Politehnica of Bucharest, Scientific Bulletin., Series C. – 2023. – Vol. 85 (3). – pp. 319-334.
2. BP I. W. K. D., Anggraini D. A Development of Modern Web Application Frontend Structures Using Micro Frontends //International Research Journal of Advanced Engineering and Science. – 2022. – Vol. 7 (1). – pp. 149-155.
3. Nikulina O., Khatsko K. Method of converting the monolithic architecture of a front-end application to microfrontends //Bulletin of National Technical University» KhPI». Series: System Analysis, Control and Information Technologies. – 2023. – Vol. 2 (10). – pp. 79-84.
4. Auer F. et al. From monolithic systems to Microservices: An assessment framework //Information and Software Technology. – 2021. – Vol. 137. – pp. 1-8.
5. Peltonen S., Mezzalira L., Taibi D. Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review //Information and Software Technology. – 2021. – Vol. 136. – pp. 1-9.
6. Taibi D., Mezzalira L. Micro-frontends: Principles, implementations, and pitfalls //ACM SIGSOFT Software Engineering Notes. – 2022. – Vol. 47 (4). – pp. 25-29.
7. Hidayat D. C., Atmaja I. K. J., Sarasvananda I. B. G. Analysis and Comparison of Micro Frontend and Monolithic Architecture for Web Applications //JurnalGalaksi. – 2024. – Vol. 1 (2). – pp. 92-100.
8. Kunštnár V., Podhorský P. Micro frontend architecture //2024 Zooming Innovation in Consumer Technologies Conference (ZINC). – IEEE, 2024. – pp. 124-129.
9. Gashi E. et al. The advantages of Micro-Frontend architecture for developing web application //2024 13th Mediterranean Conference on Embedded Computing (MECO). – IEEE, 2024. – pp. 1-5.
10. Savani N. The future of web development: An in-depth analysis of micro-frontend approaches //International Journal of Computer Trends and Technology. – 2023. – Vol. 71 (11). – pp. 65-69.

Citation: Pavel Olegovich Alekseev, "Using Micro Frontends for Modular Architecture of Web Applications", Universal Library of Engineering Technology, 2024; 1(2): 35-41. DOI: https://doi.org/10.70315/uloap.ulete.2024.0102006.