World Journal of Advanced Research and Reviews

eISSN: 2581-9615 CODEN (USA): WJARAI

Cross Ref DOI: 10.30574/wjarr

Journal homepage: https://wjarr.com/

(Review Article)

Dependency Management Strategies in Scalable Monorepositories: Analysis and Resolution of Version Conflicts

Kostadin Almishev *

Senior Information Systems Developer.

World Journal of Advanced Research and Reviews, 2025, 28(01), 648-655

Publication history: Received on 23 August 2025; revised on 06 October 2025; accepted on 09 October 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.28.1.3468

Abstract

Against the backdrop of the expansion of microservice architectures and Platform Engineering practices, monorepositories serve as a key mechanism for standardizing the software development lifecycle. At the same time, their growth exacerbates the phenomenon whereby version incompatibilities block the work of multiple teams. This study aims to systematize and analyze approaches to dependency management in scalable monorepositories in order to develop a holistic methodology for preventing and resolving version conflicts. The methodological basis includes a systematic review of academic publications, content analysis of technical documentation, and a comparative examination of industry reports. The results demonstrate the limited preventive effectiveness of semantic versioning (SemVer) with respect to compatibility errors and establish a taxonomy of management strategies: proactive (centralized version control), reactive (dependency harmonization), and automated (use of intelligent build systems). Case studies confirm that tool selection should correlate with the scale of development, implying an evolutionary transition from basic orchestrators to industrial-grade build tools such as Bazel. It is concluded that high effectiveness in dependency management is achieved through the synergy of organizational regulation, conflict resolution procedures, and the use of intelligent build systems with fine, granular analysis of the dependency graph. The practical significance of the work lies in providing architects and platform teams with a scientifically verified foundation for decision-making in the design and operation of large-scale software systems.

Keywords: Monorepository; Dependency Management; Version Conflict; Platform Engineering; Build Systems; Software Engineering; Large-Scale Development; Semantic Versioning; Nx; Bazel

1. Introduction

In contemporary software engineering practice, there is a pronounced trend toward enlarging the code base through monorepositories — single repositories in a version control system in which numerous interrelated yet autonomous components coexist [1]. This organizational-architectural choice dovetails naturally with the rise of the Platform Engineering paradigm: according to Gartner, by 2026, 80% of large enterprises will establish specialized platform teams responsible for the design and evolution of internal developer platforms (IDP) [4]. The goal-setting for such platforms is the sustained increase of engineering teams' productivity through the standardization and automation of workflows, the reduction of cognitive load, and the introduction of self-service mechanisms [5]. In this logic, the monorepository acts as a natural foundational layer: it creates a single plane for the centralized application and observability of CI/CD pipelines, security policies, static analysis procedures and — crucially — for coherent dependency management [6, 7].

Empirical observations for 2024 demonstrate the maturity of the monorepository tooling ecosystem and its broad penetration into professional software development. According to the State of JavaScript 2024 survey, solutions such as pnpm (3 508 respondents), npm Workspaces (2 438), Nx (1 917), and Turborepo (1 412) are used steadily to

* Corresponding author: Kostadin Almishev

maintain complex project landscapes [8]. This dynamic reflects the industry's institutionalized drive toward governability and efficiency against the backdrop of the growing systemic complexity of software products.

At the same time, scale effects make the approach's reverse side visible: although a monorepository facilitates cross-project modifications and strengthens team cooperation, it simultaneously exacerbates the problem of dependency management. Survey results identify the leading pain points: laborious configuration (137 mentions), excessive complexity (108), as well as difficulties with package managers and dependencies (107 and 81 mentions respectively) [8]. In a monorepository these risks intensify nonlinearly: a version conflict of the same external library, transitively used by dozens of internal packages, can halt build, testing, and deployment, effectively blocking the operational contour of the entire organization [9]. Consequently, ineffective dependency management nullifies the key benefits of the platform approach and transforms from a local engineering task into a factor of strategic vulnerability.

Contemporary research discourse typically either conceptualizes dependency management within classical poly-repository practices or examines individual tooling solutions in isolation. At the same time, a systematic overview of the phenomenon of version conflicts specifically in scalable monorepositories remains underexplored, as a multifaceted problem at the intersection of organizational processes (e.g., Large-Scale Agile), architectural approaches (Platform Engineering), and tooling ecosystems (Build Systems).

The aim of this study is to systematize and analytically interpret existing dependency management strategies in scalable monorepositories in order to formulate an integrated approach to the prevention and resolution of version conflicts.

The working hypothesis is that the effectiveness of dependency management in monorepositories is determined not by the choice of a single tool but by a balanced configuration of three complementary components: proactive architectural-organizational policies, reactive conflict-resolution mechanisms, and the use of smart build systems with granular dependency analysis and distributed caching.

The scientific novelty lies in the development of a taxonomy of dependency management strategies specifically adapted to the specifics of monorepositories and reflecting the inseparable linkage between the technical and organizational dimensions of the problem under consideration.

2. Materials and methods

The study relies on an interdisciplinary approach that combines several complementary methods to ensure comprehensive coverage and objective analysis of the research problem. The methodological core is a systematic literature review aimed at identifying, critically appraising, and synthesizing relevant scholarly publications. Within this approach, peer-reviewed articles indexed in Scopus and Web of Science, papers from leading software engineering conferences (IEEE/ACM), as well as preprints and dissertations were examined sequentially. The application of this approach made it possible to construct a theoretical framework for understanding the fundamental aspects of dependency management in large-scale projects.

To analyze contemporary tools, practices, and architectural patterns, content analysis was employed. The technical documentation of key build systems (Bazel, Nx, Turborepo), industry publications, as well as materials from practical case studies and discussions in professional communities were examined. This method provided an in-depth examination of the applied aspects of implementing dependency management strategies.

To document current trends, the diffusion of technologies, and the most acute problem areas for developers, a comparative analysis of data from authoritative industry reports was conducted. The principal empirical sources comprised Gartner analytical materials and the results of the global State of JavaScript survey.

The source base covers more than 20 carefully selected sources published in recent years, which ensures the relevance and validity of the conclusions. The corpus is divided into two types:

Academic sources: scholarly articles and conference papers that form the theoretical foundation of the study; they address, in particular, technical dependencies in scalable Agile development, the empirics of version conflicts, and the systematization of knowledge on monorepositories.

Industry reports and technical reviews: analytics and survey data providing an empirical picture of the state of the industry, the dynamics of Platform Engineering, the popularity of tools, and practical challenges.

This combination of methods made it possible not only to consolidate existing theoretical perspectives but also to compare them with observed practice, thereby ensuring the high theoretical-methodological and applied value of the results obtained.

3. Results and discussion

The problem of dependency management is by no means limited to the specifics of monorepositories; it stems from the fundamental challenges of industrial software system development. Research on Large-Scale Agile consistently documents a set of five interdependent nodes: flawed planning, conflicting prioritization, insufficient knowledge circulation, code quality degradation, and integration difficulties [10-12]. Taken together, they shape managerial decisions made with a superficial grasp of the technical context, leading to suboptimal decomposition and work allocation, which generates unforeseen inter-team technical dependencies. These, in turn, derail plans, intensify competition among priorities, and trigger unplanned communications between teams. Time and information deficits are translated into code deterioration, complicating subsequent integration and undermining the quality of future planning, thereby closing the loop (see Fig. 1).

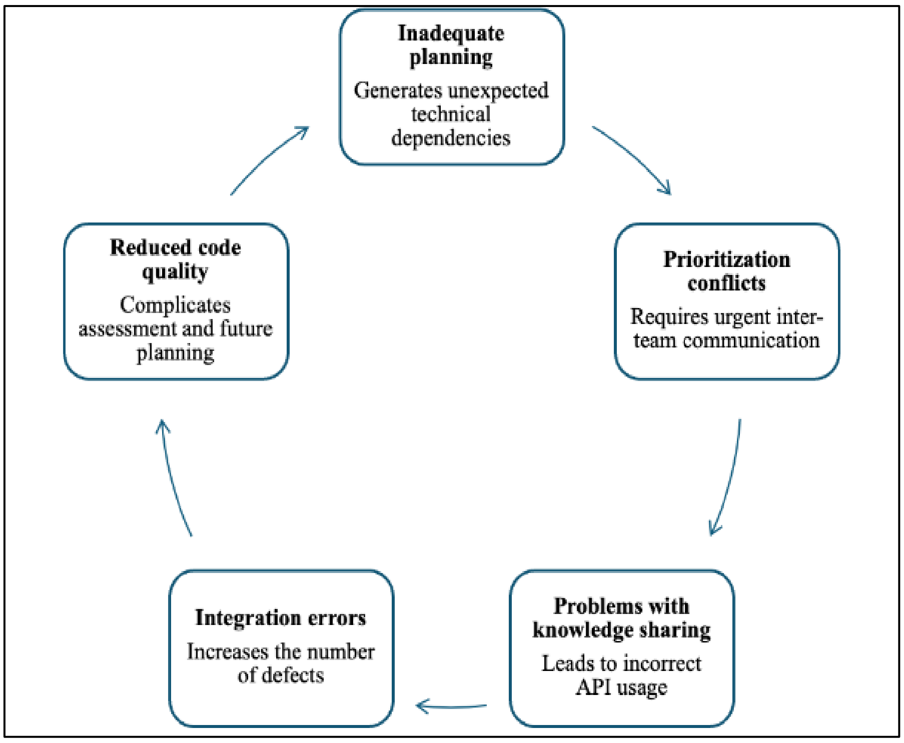
Inadequate planning
Generates unexpected technical dependencies
Prioritization conflicts
Requires urgent inter-team communication
Problems with knowledge sharing
Leads to incorrect API usage
Integration errors
Increases the number of defects
Reduced code quality
Complicates assessment and future planning

Figure 1 Conceptual model of dependency problems (compiled by the author based on [10-12; 13, 15])

In the context of a monorepository, this closed loop is not merely reproduced but takes on a hypertrophied form. Tight coupling of modules, a shared dependency matrix, and a single version control line turn a local failure by one team into a systemic effect that rapidly engulfs the entire codebase [2, 3]. Empirical data support this logic: according to the State of JavaScript 2024 survey, the dominant difficulties in working with monorepositories include Configuration, Excessive complexity, and Package management, which directly indicates vulnerabilities in integrity and controllability in a large-scale environment [8, 14].

The primary cause of most technical failures is version collisions: they arise when a project directly or transitively pulls in several incompatible editions of the same external library [10]. The industry-standard semantic versioning scheme (SemVer) with the MAJOR.MINOR.PATCH triad does not, in practice, provide the required compatibility guarantees [20]. An empirical analysis conducted at Delft University of Technology on 124 pull requests demonstrated that relying solely on syntactic numbering rules is a methodological trap: roughly 80% of runtime errors were due to direct incompatibility, where a loaded older version of a library lacked functionality expected by code compiled against a newer edition. SemVer does not protect against such situations, and in a number of cases violations of backward compatibility were observed despite formal adherence to its prescriptions [10, 16].

This conclusion is especially significant for monorepositories. In an environment where hundreds of internal packages share the same base library (for example, React or Jackson), automatically bumping even the minor number under SemVer rules (say, from 16.8.0 to 16.9.0) can trigger a cascading failure of the entire system. A small yet disruptive API change, not reflected in the version, is sufficient to break the operation of dozens of dependent modules. Consequently, viable approaches to dependency management should shift the emphasis from formal version control to verification of actual compatibility through comprehensive integration testing that covers all affected segments of the repository.

An analysis of academic literature and industry practice makes it possible to differentiate approaches to resolving version conflicts in monorepositories into three classes: proactive, reactive, and automation-based.

Proactive strategies. Their goal is to minimize the likelihood of conflicts through normative organizational and technical regimes and architectural constraints.

Centralized version management: the strictest model is one in which all projects within a monorepository are forcibly aligned to unified versions of external dependencies. This is achieved by hoisting dependencies to the repository root and declaring them in a single instance [21]. The main advantage is strong consistency and the elimination of duplicate packages. The principal difficulty is the need to update all dependent codebases synchronously when migrating, for example, to a new major version of a framework; in large repositories such an operation becomes extremely resource-intensive, effectively paralyzing development for weeks [21].

Strict versioning policies and automation: a softer but effective line that unifies proven practices. The CODEOWNERS configuration in a version control system makes it unambiguous to assign responsibility for modules and their dependencies to specific teams or engineers, improving manageability [22]. Pre-commit hooks that track attempts to add dependencies bypassing regulations prevent uncontrolled version fragmentation [21]. Complementing the approach is automatic changelog generation based on semantic commits (for example, feat:, fix:, BREAKING CHANGE:), which increases transparency and facilitates assessing the impact of updates [20].

Turning to reactive strategies, they take effect post factum — when the conflict has already manifested — and aim at its prompt removal. Version harmonization in this case is the most typical practice, accounting, according to the study, for 67,7% of conflict resolution cases [10]. The essence is a manual or semi-automatic analysis of the dependency graph in search of a single library version compatible with all affected components, followed by hard locking of this version (version pinning) in the configuration. With preventive detection, instead of waiting for runtime failures, tools embedded in the build process are used. Thus, Maven Enforcer with the dependencyConvergence rule identifies multiple versions of the same dependency at compile time, immediately aborts the build, and signals the developer about the problem. As a result, the conflict is eliminated before it enters the artifact [10].

In automated strategies, the emphasis is on specialized build systems designed for large-scale monorepositories. Tools such as Nx, Turborepo, and Bazel go beyond package management: they construct a complete dependency graph of the repository and thereby implement a critically important optimization — incremental builds and tests. When code changes, graph analysis makes it possible to run the build and verification only for those projects that are directly modified or transitively depend on them (the affected projects principle) [17]. This drastically reduces the duration of CI/CD pipelines and is a necessary condition for maintaining high performance in scalable monorepositories. The popularity of such tools continues to grow, which is confirmed by survey data (see Fig. 2).

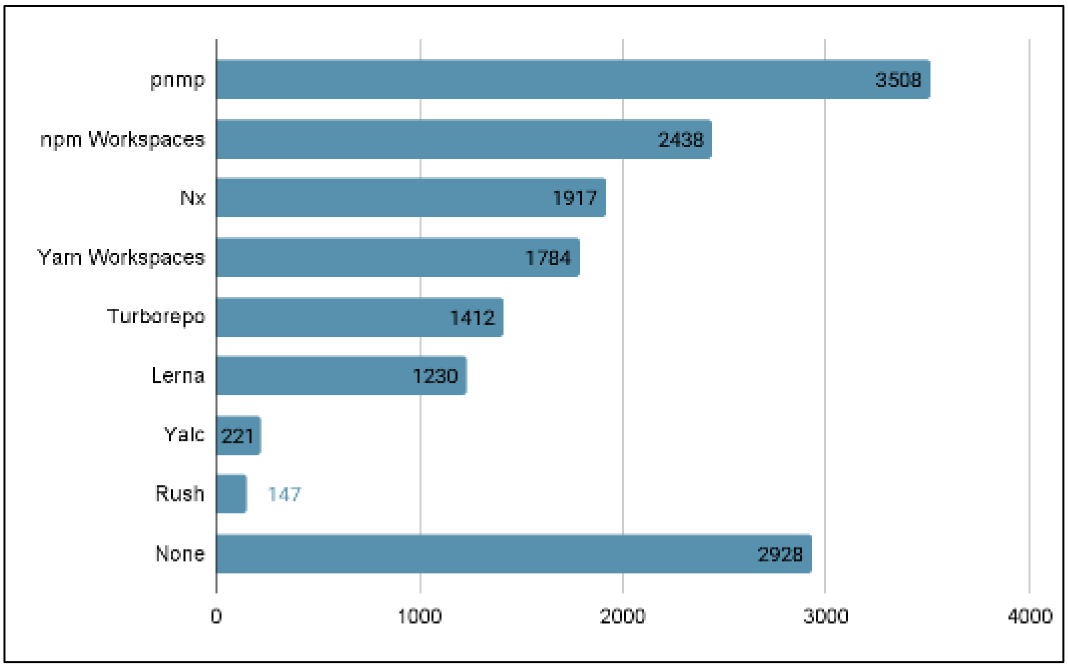

Figure 2 The prevalence of instruments for monorepositories in the professional environment (compiled by the author on the basis of [8])

The correct selection of the tooling stack is a key determinant of the successful operation of monorepositories, as it shapes the CI/CD throughput profile, the scaling limits, and the quality of the developer experience. Current solutions are appropriately typologized by gradations of complexity and computational capacity, as clearly reflected in Table 1.

Table 1 Comparative analysis of key build systems for monorepositories (compiled by the author based on [24]).

Criterion	**Lerna & Yarn/NPM Workspaces**	**Nx & Turborepo**	**Bazel**
Granularity of analysis	Package level	Package and file level (source code analysis)	File level (hermetic, reproducible builds)
Caching strategy	Absent (depends on the package manager)	Local and distributed (remote) caching	Local and distributed caching, remote execution of builds
Language support	Predominantly JavaScript/TypeScript	JavaScript/TypeScript, extensible via plugins (Go, .NET)	Multilingual (Java, C++, Go, Python, JS/TS, etc.)
Entry threshold	Low	Medium	High
Ideal scenario	Small and medium JS/TS projects, package management	Large JS/TS-oriented projects, web applications	Very large, multilingual projects with high requirements for performance and reproducibility

Based on the data presented in Table 1, it is worth noting that Lerna and Yarn/NPM Workspaces are base-layer tools whose primary function is package orchestration, version management, and the application of hoisting. Their value lies in minimal adoption overhead and configuration simplicity, which makes them a reasonable starting option. However, as the monorepository grows, limitations emerge: the absence of built-in intelligent caching and fine-grained dependency analysis leads to redundant rebuilds and repeated test runs, which degrades pipeline performance [24].

Nx and Turborepo are next-generation solutions for monorepositories, distinguished by advanced caching strategies and detailed dependency-graph analysis. They consider not only inter-package relationships but also changes in source code at the artifact level, thereby precisely determining the impact scope of each commit. As a result, only the strictly necessary subset of tasks is rebuilt and reused, which substantially accelerates CI/CD. Support for Remote Caching enables sharing build results among developers and CI agents, further increasing team efficiency [17].

Bazel (developed at Google) is the most powerful, yet at the same time the most labor-intensive to master, industrial-grade tool. It provides hermetic, reproducible, and incremental builds with file-level granularity: final artifacts do not depend on the execution environment, and rebuilds affect only those components whose sources have changed. Originally oriented toward polyglot codebases and operation at the scale of Google's monorepository, Bazel requires substantial engineering investment and deep expertise during adoption [25].

Analysis of practice demonstrates an evolutionary trajectory of tooling development; a rational choice is determined by the current scale and maturity of the project.

Thus, within case 1 - Civo (transition to Dagger), it should be noted that the cloud provider consolidated code into a monorepository to simplify dependency management and improve collaborative development. As the codebase grew, typical CI/CD pipelines swelled to ~30 minutes. Integrating Dagger — programmable pipelines with powerful caching — reduced the average time to 5 minutes. The system learned to compute precisely which parts of the monorepository require rebuilding, and even to select relevant Go versions for individual projects dynamically [23]. This example shows how intelligent build infrastructure eliminates performance bottlenecks.

Case 2: Cognite (migration from Lerna/Yarn to Bazel). As the monorepository expanded and the number of teams grew, the Lerna/Yarn stack ceased to provide the required scalability: CI time reached 35 minutes. The company made a strategic decision to switch to Bazel. Despite high complexity and costs (one developer spent six weeks on the rollout), the effect proved impressive: the average build time dropped to 3 minutes. Owing to fine-grained caching and precise dependency analysis, adding new services now only minimally affects the overall CI/CD duration [26, 27].

The presented cases reveal a stable pattern: it is insufficient for organizations to select a tool opportunistically for current tasks — it is necessary to build a roadmap for the incremental development of the technology stack. As the project grows and becomes more complex, a tipping point arises when the cumulative costs of long builds exceed the expenses of implementing and operating a more complex but high-performance architecture.

The next turn in dependency management is associated with the widespread incorporation of artificial intelligence. The AI-Augmented Development direction, noted by Gartner as one of the key trends of 2024, is aimed at radically increasing developer productivity [4]. AI tools are already penetrating IDEs and CI/CD pipelines, removing routine through automation of code generation, refactoring, test writing, and debugging [28].

The same logic extends to working with dependencies. In practice, approaches are used in which AI assumes responsibility for resolving version collisions: pipelines using large language models (LLM), for example Claude, scan all package.json files in the monorepository, identify potential conflicts, and automatically create pull requests with recommendations for harmonized versions [2, 9]. IDE assistants (for example, GitHub Copilot) obtain a qualitatively more complete context in a monorepository, which makes it possible to deliver targeted advice on dependency updates, forecast their impact on adjacent subsystems, and propose refactoring options for new APIs [18, 19].

In the future, one can expect the emergence of next-level tools. They will go beyond fixing syntactic incompatibilities and perform semantic analysis of inter-version changes (based on changelogs, commits, and source code). Using such signals, AI will be able to assess specific risks for target code fragments in the monorepository and formulate optimal migration strategies — up to and including automatic generation of adapting inserts to overcome breaking changes.

4. Conclusion

The conducted analysis demonstrates that dependency management is one of the key and least trivial tasks when scaling monorepositories. The escalation of version conflicts under conditions of high artifact coupling requires not the ad hoc selection of a tool, but a holistic, multi-level strategy. Key propositions of the study are formulated as follows:

The problem is systemic in nature. Dependency conflicts in monorepositories stem from the fundamental complexities of large-scale development — deficiencies in planning and communication, while the monorepository architectural model only amplifies and makes these organizational shortcomings more explicit.

Semantic Versioning (SemVer) is not a sufficient condition. Empirical observations show that the formal correctness of version numbers does not guarantee actual component compatibility. Effective practices complement syntactic rules with semantic validation through full-scale automated testing.

Multi-level management is required. An effective strategy combines proactive measures (centralized policies and automated controls), reactive mechanisms (a procedure for version harmonization), and, which is critical for scalability, automated solutions based on intelligent build systems.

Thus, the author's hypothesis is confirmed: successful dependency management in scalable monorepositories is achieved through the synergy of three components — proactive management, reactive settlement, and the prioritized use of intelligent build systems (Nx, Turborepo, Bazel), which minimize the effects of high coupling through granular graph analysis and advanced caching.

The practical significance of the work lies in the fact that the obtained results can serve as a foundation for software architects, technical leads, and platform teams in strategic decision-making. The proposed taxonomy of strategies and the comparative analysis of tooling form a scientifically grounded basis for selecting and implementing approaches commensurate with the scale and specifics of projects. Reflection on the evolution of tools — from simple orchestrators to industrial-grade build systems — enables the systematic development of technical infrastructure, reducing risks and increasing the long-term efficiency of development in large software ecosystems.

References

[1] Lando B., Hasselbring W. Toward Bundler-Independent Module Federations: Enabling Typed Micro-Frontend Architectures //2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C). – IEEE, 2025. – pp. 36-40.

[2] Labba Awwabi, Siti Rochimah. Evaluating Development Velocity: A Systematic Comparison of Monorepo and Polyrepo Architectures. International Journal of Innovative Science and Research Technology (IJISRT). - 2025. - Vol. 10(2) - pp. 1652–1659. https://doi.org/10.5281/zenodo.14965864.

[3] Brousse N. The issue of monorepo and polyrepo in large enterprises //Companion proceedings of the 3rd international conference on the art, science, and engineering of programming. – 2019. – pp 1-4. https://doi.org/10.1145/3328433.3328435.

[4] The Software Engineering Trends Shaping 2024 [Electronic resource]. - Access mode: https://www.axented.com/blog-posts/top-trends-for-software-engineering-in-2024-according-to-gartner (date of access: 20.08.2025).

[5] 2024 State of DevOps Report: The Evolution of Platform Engineering [Electronic resource]. - Access mode: https://www.puppet.com/system/files/2025-03/pup-2024-sodor-report.pdf (date of access: 22.08.2025).

[6] Automate Servers, Networks, and Edge Devices [Electronic resource]. - Access mode: https://www.puppet.com/products/puppet-edge-management (date of access: 23.08.2025).

[7] Costs exposed: Monorepo vs. multirepo [Electronic resource]. - Access mode: https://jmmv.dev/2023/08/costs-exposed-monorepo-multirepo.html (date of access: 25.08.2025).

[8] Monorepo Tools - State of JavaScript 2024 [Electronic resource]. - Access mode: https://2024.stateofjs.com/so-SO/libraries/monorepo_tools/ (date of access: 29.08.2025).

[9] Lando B., Hasselbring W. Toward Bundler-Independent Module Federations: Enabling Typed Micro-Frontend Architectures //2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C). – IEEE, 2025. – pp. 36-40.

[10] Mihaila V. V. An Empirical Study of Version Conflicts in Maven-Based Java Projects. – 2025. - pp.1-11.

[11] Saeeda H., Ahmad M. O., Gustavsson T. Challenges in large-scale agile software development projects //Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing. – 2023. – pp. 1030-1037.

[12] Vedal H. et al. Managing dependencies in large-scale agile //International Conference on Agile Software Development. – Cham : Springer International Publishing, 2021. – pp. 52-61.

[13] Avuthu Y. R. Scaling devops: Challenges in managing large and distributed teams //ESP Journal of Engineering Technology Advancements. – 2022. – Vol. 2 (1). – pp. 100-106.

[14] Tkalich A., Klotins E., Moe N. B. Identifying Critical Dependencies in Large-Scale Continuous Software Engineering //arXiv preprint arXiv:2504.21437. – 2025. - pp.1-6.

[15] Shakikhanli U., Bilicki V. Optimizing branching strategies in Mono-and Multi-repository environments: A comprehensive analysis //Computer Assisted Methods in Engineering and Science. – 2024. – Vol. 31 (1). – pp. 81-111.

[16] McIntosh S. et al. Using Reinforcement Learning to Sustain the Performance of Version Control Repositories //2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). – IEEE, 2025. – pp. 116-120.

[17] Monorepo Tooling in 2025: A Comprehensive Guide [Electronic resource]. - Access mode: https://www.wisp.blog/blog/monorepo-tooling-in-2025-a-comprehensive-guide (date of access: 30.08.2025).

[18] JavaScript Monorepos in 2024: Legit or Sus? [Electronic resource]. - Access mode: https://keyholesoftware.com/javascript-monorepos-in-2024-legit-or-sus/ (date of access: 03.09.2025).

[19] Monorepos Are Back — And AI Is Driving the Comeback [Electronic resource]. - Access mode: https://medium.com/@dani.garcia.jimenez/monorepos-are-back-and-ai-is-driving-the-comeback-f4abbb7bb55f (date of access: 05.09.2025).

[20] Best Practices for Versioning in Monorepos Using Gitflow [Electronic resource]. - Access mode: https://moldstud.com/articles/p-best-practices-for-versioning-in-monorepos-using-gitflow (date of access: 06.09.2025).

[21] Linkola L. Design and implementation of modular frontend architecture on existing application //Information Technology. – 2021. - pp.1-47.

[22] Common pitfalls when adopting a monorepo (and how to avoid them) [Electronic resource]. - Access mode: https://graphite.dev/guides/monorepo-pitfalls-guide (date of access: 10.09.2025).

[23] Adopting a Monorepo Strategy: Civo's Experience [Electronic resource]. - Access mode: https://dagger.io/blog/adopting-monorepo-strategy (date of access: 10.09.2025).

[24] LERNA VS YARN WORKSPACES: WHAT ARE THE DIFFERENCES? [Electronic resource]. - Access mode: https://www.startechup.com/blog/lerna-vs-yarn-workspaces/ (date of access: 13.09.2025).

[25] Comparing Bazel, Lerna, Nx, and Pants [Electronic resource]. - Access mode: https://graphite.dev/guides/monorepo-tooling-comparison (date of access: 13.09.2025).

[26] Monorepo Tools: A Comprehensive Comparison [Electronic resource]. - Access mode: https://graphite.dev/guides/monorepo-tools-a-comprehensive-comparison (date of access: 17.09.2025).

[27] Migrating from Lerna with Yarn Workspaces to Bazel [Electronic resource]. - Access mode: https://medium.com/cognite/migrating-from-lerna-with-yarn-workspaces-to-bazel-af8abb962940 (date of access: 19.09.2025).

[28] Turning the 2024 State of DevOps into your 2025 Playbook for DevOps Excellence [Electronic resource]. - Access mode: https://www.cortex.io/post/2025-playbook-for-devops-excellence (date of access: 23.09.2025).