

MONOREPOSITORY-DRIVEN DESIGN OF INFORMATION SYSTEMS AND SOFTWARE COMPLEXES

MAIDANIUK V, KHOSHABA O (pzmag2022@gmail.com)

Vinnitsia National Technical University (Ukraine)

Abstract. Large-scale information systems (IS) and software complexes increasingly integrate heterogeneous services, interfaces and data pipelines, yet fragmented repositories and brittle cross-team processes often constrain their evolution. This paper presents a monorepository-driven approach to the design of IS and software complexes, treating the repository, the build system, and the release process as a unified socio-technical architecture. We formulate the design problem, derive a set of architectural and organisational constraints from first principles and prior art, and synthesise a method that aligns dependency governance, hermetic builds and continuous delivery with cross-team change management. The contribution is a design framework with actionable rules for repository topology, dependency semantics and CI orchestration, accompanied by an evaluation protocol and quality indicators suitable for pilots in institutional environments. While our focus is architectural, we outline how the approach yields measurable improvements in build determinism, change-coupling visibility and time-to-integration, and we discuss threats to validity and portability to polyglot toolchains.

Keywords: monorepository, information systems design, software complexes, build systems, dependency management, CI/CD, change management, socio-technical architecture

Problem statement. Contemporary IS and software complexes are assembled from many services and libraries that evolve asynchronously. Multi-repository layouts amplify coordination overheads, encourage version skew and impede systematic refactoring. Empirically, teams face three recurrent failure modes: hidden change coupling across repositories, non-deterministic builds due to environment drift, and elongated integration cycles in CI/CD pipelines. The design challenge is to provide a repository-centred architecture that exposes coupling early, enforces reproducibility end-to-end, and enables consistent, organisation-wide refactoring without sacrificing the autonomy of product teams. Existing accounts from industry illustrate feasibility at a considerable scale but offer limited prescriptive guidance for medium-sized institutions operating polyglot stacks and mixed workloads [1], [2].

Objectives. The work pursues four objectives: first, to formalise repository topology and dependency semantics that minimise accidental complexity; second, to specify a build and test substrate that guarantees hermeticity and cacheable execution across languages; third, to align continuous integration with change impact analysis so that cross-project changes are safe by construction; and fourth, to define operational governance for branching, code ownership and review that scales across teams while preserving accountability [1]-[6].

Essence of the method. The proposed method views the monorepository as an architectural artefact rather than a mere storage choice. At the structural level, we model all buildable artefacts - applications, libraries, configuration and documentation - as first-class graph nodes with explicit, acyclic dependencies. Interfaces across bounded contexts are stabilised via versioned API contracts stored in-repo; transitive exposure of internal symbols is disallowed to reduce ripple effects. At the tooling level, we favour a hermetic build substrate with remote execution and content-addressable caching to ensure determinism and parallelism across a heterogeneous estate [2], [3]. Build targets declare all inputs, including toolchains; external dependencies are vendored or pinned via lockfiles to eliminate network variability. At the process level, we replace default trunk-based development with a “change set” protocol: multi-project changes are expressed as atomic, reviewable units that traverse a single CI pipeline capable of selective builds, affected-graph detection, and test partitioning [4]-[6]. Code ownership is expressed as fine-grained metadata attached to paths;

reviewers are assigned automatically on the dependency cut. Release engineering adopts “release-as-a-set” for tightly coupled components and “contract-first” publishing for decoupled ones; both modes share provenance and SBOM generation to support compliance. Finally, we integrate telemetry from the build substrate into the design loop: cache hit ratios, flake rates and critical path lengths function as design feedback rather than pure operational metrics [1]-[6], [8].

Research contribution. The contribution is threefold. First, we articulate design rules that make cross-team refactoring tractable: explicit interfaces, prohibition of transitive leaks, and enforcement of no-cycle policies at the repository root, with automated checks that block violations at review time. Second, we provide an implementation-agnostic evaluation protocol comprising quality indicators for reproducibility, integration latency and change-coupling visibility that can be adopted in pilots without prescribing a specific vendor tool [2]-[6]. Third, we report a portability discussion that maps the rules to polyglot stacks typical of institutional IS: JVM/TypeScript/Python ecosystems with both frontend and data workflows, highlighting where tool-specific affordances differ (e.g., remote execution vs task graph scheduling) while preserving the invariants of the design [3]-[6], [8].

Evaluation protocol (pilot-ready). To enable evidence-based adoption, we define an evaluation protocol suitable for a three-month pilot in an institutional setting. The protocol specifies three primary indicators: time-to-integration (the median wall-clock time from change submission to mainline acceptance), build determinism (the proportion of bit-identical rebuilds under controlled cache conditions), and change-coupling visibility (the share of merged change sets that span more than one bounded context with all owners participating in review). Secondary indicators include cache hit ratio on CI, test flakiness rate and the size of the critical path in the build graph. Measurement occurs continuously via CI telemetry and weekly snapshots. Statistical treatment relies on non-parametric tests due to skew typical of CI timings; practical significance is assessed through relative effect sizes and bootstrap confidence intervals [1]-[6], [8]. This protocol is presented to make replication straightforward, even when institutional constraints preclude the sharing of raw pipelines.

Results and practical observations. Although the present paper centres on design rather than exhaustive experimentation, the method has been trial-planned with explicit acceptance thresholds that are realistic for medium-sized engineering groups. For adoption to be considered successful, we propose targets of a marked reduction in median time-to-integration relative to a multi-repository baseline, sustained high determinism in controlled rebuilds, and demonstrable improvement in cross-team review coverage on change sets affecting shared libraries. These thresholds and their operationalisation are aligned with the prior art on monorepositories at scale and the documented capabilities of modern build and scheduling substrates [1]-[6]. We emphasise that numerical outcomes are environment-dependent; therefore, organisations should publish both central tendencies and dispersion to avoid over-generalisation.

Discussion and threats to validity. The method assumes institutional willingness to standardise toolchains and to invest in a robust CI substrate; under-provisioned remote execution, intermittent artefact storage or unmanaged third-party dependencies will limit benefits. Polyglot systems without mature rulesets may experience uneven performance until per-language build rules reach parity. Socio-technical resistance is non-trivial: code ownership and review discipline must be enforced consistently, or change-set atomicity degrades in practice. Differences in regulatory and security contexts constrain external validity; however, the evaluation protocol is designed to expose such constraints early and to support incremental adoption [1], [2], [7], [8].

Conclusions. A monorepository-driven approach can be framed as a coherent design for information systems and software complexes when repository topology, dependency semantics, build determinism and change governance are treated as interlocking parts of a single architecture. The method presented here contributes explicit rules, an implementation-agnostic tool mapping and a pilot-ready evaluation protocol. Future work includes a controlled study across multiple institutions and an extension to data-intensive workflows where lineage and privacy constraints interact with build determinism. The broader implication is that repository design is not merely an

operational choice but a primary architectural decision that shapes the evolvability of institutional information systems [1]-[8].

References:

- [1] R. Potvin and J. Levenberg, “Why Google Stores Billions of Lines of Code in a Single Repository,” *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016. [Online]. Available: <https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository> [Accessed: 19 Oct. 2025].
- [2] Bazel, “Bazel — Fast, scalable, multi-language and extensible build system,” Documentation, 2025. [Online]. Available: <https://bazel.build> [Accessed: 19 Oct. 2025].
- [3] Nx (Nrwl), “Nx: Next generation build system with first-class monorepo support,” Documentation, 2025. [Online]. Available: <https://nx.dev> [Accessed: 19 Oct. 2025].
- [4] Pantsbuild, “Pants — A fast, scalable Python-friendly build system,” Documentation, 2025. [Online]. Available: <https://www.pantsbuild.org> [Accessed: 19 Oct. 2025].
- [5] Vercel, “Turborepo — High-performance build system for JavaScript and TypeScript codebases,” Documentation, 2025. [Online]. Available: <https://turbo.build> [Accessed: 19 Oct. 2025].
- [6] M. Gligoric, L. Eloussi, and D. Marinov, “Practical Regression Test Selection with Dynamic File Dependencies,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA ’15)*, 2015, pp. 211–222. [Online]. Available: <https://doi.org/10.1145/2771783.2771784> [Accessed: 19 Oct. 2025].
- [7] J. Kerzic and J. Yuen, “Hermetic Builds for Reproducible Systems: Principles and Practice,” *IEEE Software*, vol. 39, no. 4, pp. 33–40, 2022. [Online]. Available: <https://doi.org/10.1109/MS.2021.3139057> [Accessed: 19 Oct. 2025].
- [8] ThoughtWorks, “Monorepo,” *Technology Radar*, 2024. [Online]. Available: <https://www.thoughtworks.com/radar/techniques/monorepo> [Accessed: 19 Oct. 2025].