Architectural Analysis of Unified Design Systems for Global Web Platforms Scaling Across 180 Countries

Mykhailo Nykoliuk
Lead Frontend Engineer, Ajax Systems
5 Semena Skliarenka St.
Kyiv, 04073, Ukraine

ABSTRACT
Global web platforms face a scaling problem that is organisational as much as technical: as the number of served markets grows, the cost of publishing and maintaining a single localized page grows with it. This study examines that problem through an industrial case study of a corporate platform operating across 180 countries, where a legacy WordPress installation was replaced by a decoupled architecture combining Next.js, a headless content management system, and a centralized design system. The research applies system analysis rather than code-level benchmarking. Its object of evaluation is the architecture together with the deployment and localization processes it supports, and operational indicators recorded across five quarters serve as evidence. Median time to publish a localized page fell by 25 percent and engaged sessions per visitor rose by 40 percent, while published output rose by 176 percent at an engineering budget that changed by 1.8 percent. Decomposition of publication time shows that the architecture addressed 42.5 percent of the baseline elapsed time and removed 58.8 percent of that addressable portion. The contribution is a transferable blueprint for organisations that must standardize interface production across many national markets without fragmenting engineering effort across regional teams.

General Terms
Software Architecture, Systems Analysis, Design, Performance, Measurement, Management, Standardization, Human Factors.

Keywords
Industrial case study; design system; frontend architecture; enterprise scalability; web modernization; headless CMS; component reuse; localization.

1. INTRODUCTION

1.1 Background
International expansion has changed what a corporate web platform must do. A site that once served one market with a few landing pages now operates as a distribution network: dozens of languages, regional legal notices, local contact and payment conventions, and marketing calendars that rarely align. Each new market adds a family of pages, and every page must stay consistent with the rest. Growth of this kind is linear in appearance and superlinear in cost.

The economics turn unfavourable when the underlying system is monolithic. Maintenance of legacy enterprise applications absorbs a substantial share of IT budgets and leaves limited capacity for new capability, the pattern modular design was formulated to correct [1]. Modernization is therefore not only technology selection. Analysis of programmes across financial services, healthcare, and contract management shows that outcomes depend on assessment methodology, data migration planning, and organisational change management in roughly equal measure [2].

At the presentation layer the problem takes a particular form. Navigation bars, product cards, form controls, and cookie banners are conceptually shared across every market, yet in a monolithic installation they exist as templates bound to a single deployable unit. Regional teams that need a variant reimplement it. Divergence accumulates quietly, then surfaces as brand inconsistency, duplicated maintenance, and accessibility defects repaired in one locale and left standing in forty others.

1.2 Problem Statement
In the platform under study, presentation, content, and delivery were bound in one WordPress deployment. A change to a shared template required a full release cycle affecting all markets at once, which raised the risk of every change and reduced how often changes were attempted. Regional editors could not publish independently of the engineering release train, and the queue of pending localized pages lengthened as markets were added. Reviews of migration practice describe this configuration precisely: stable modernization depends on phased transition, business-aligned bounded contexts, and explicit persistence boundaries, none of which a template-coupled monolith provides [3]. Such restructuring also redefines service boundaries, data ownership, and coordination routines, and early instability is expected rather than anomalous [4].

The research problem follows. Coupling between content production and interface delivery, combined with the absence of a standardized component layer, produces deployment delays that scale with the number of served markets. The problem is architectural and managerial. It does not yield to optimization at the level of individual code paths.

From this statement, the following research tasks follow:

- To characterize the bottlenecks of the legacy stack in deployment coupling, component duplication, and localization latency;
- To formalize the target architecture and the division of responsibility between the rendering layer, the content repository, and the design system;
- To define criteria appropriate to an architectural rather than algorithmic intervention;
- To assess the operational change against those criteria and map decisions to business outcomes;
- To position the results against published evidence on

monolith-to-modular migration.

1.3 Objectives

This study evaluates the transition from a legacy WordPress stack to a decoupled architecture on Next.js and a headless content management system, through system analysis rather than code-level testing. The choice is deliberate. Where the intervention is structural, the indicators are operational: deployment frequency, time from content readiness to publication, component reuse, and downstream engagement. Benchmarking work on migration to service-based architectures establishes both the legitimacy and the limits of such indicator sets [5]. This study adopts the process-level subset, since the object of change is the production pipeline rather than the runtime hot path.

2. LITERATURE REVIEW

The literature divides along three lines: the structural progression from monolithic to modular systems, the quantification of outcomes from that progression, and the architectural choices available to organisations operating across many regions.

2.1 Evolution from Monolith to Modularity

Early treatments framed decomposition in terms of scalability, maintainability, and continuous delivery, and identified the cultural dimension alongside the technical one [6]. Systematic work has been more precise. A review of thirty-five studies produced a decomposition framework and concluded that the field remains at an early stage, with insufficient tool support and no standardized metrics, datasets, or baselines [7]. That absence of shared measurement conventions constrains what any single case study can claim, and it is acknowledged here.

Method-oriented reviews map the available techniques. Kaloudis [8] evaluated Domain-Driven Design and the Saga pattern through the Netflix and Amazon cases and proposed a phased migration path. Hassan and colleagues [9] surveyed methodologies and algorithms with attention to automation. Agaev [10] placed the modular monolith between the two poles and tied success to managed combinations of methods rather than to any single strategy.

The intermediate position deserves emphasis. Faustino and colleagues [11] measured migration effort and performance impact for a stepwise transition, treated the modular monolith as a distinct stage, and asked whether architects should complete the migration or stop there. Refactoring work favours incremental modularization through bounded context identification and dependency graph restructuring over disruptive rewrites [12]. Strategic frameworks from financial services, government, and retail describe three recurring patterns: the Strangler Fig, domain-driven decomposition, and capability-oriented refactoring [13]. Automation has advanced to pattern-based frameworks using clustering and ensemble methods to propose service boundaries [14], though comparative analyses still credit monolithic designs with advantages in simplicity [15].

2.2 Performance Metrics and Business Outcomes

The second line supplies the quantitative anchor. Wang [16] studied composable enterprise architecture built from packaged business capabilities and reported a 55 percent reduction in downtime, deployment frequency moving from quarterly to monthly, and a 60 percent decrease in time-to-market. Lakshmi and Mangalagiri [17] documented a 67 percent reduction in release cycle time and a 30 percent decrease in post-release defects across a five-year modernization of a licensing platform. In retail, composable commerce architectures produced improvements of up to 53 percent in feature deployment speed with 20 percent cost savings [18].

Runtime evidence points the same way while measuring something different. A study of migration from monolithic middleware, using 441 system interaction activities under Docker, Kubernetes, and Kafka, recorded lower message processing latency and higher throughput [19]. Controlled comparison of one application deployed monolithically and in containers, under identical stress tests, yielded quantified trade-offs rather than uniform improvement [20]. Methodological work supplies the instruments: indicator sets validated through practitioner survey [5], and decision roadmaps benchmarking monolith, service-oriented, and microservice options against performance, time-to-market, stability, and cost [21]. Delivery-oriented architecture models using domain-driven design and API-first development report faster time-to-market across their case portfolios [22].

Two cautions attach to these figures. They come from heterogeneous settings and rest on baselines rarely described in comparable terms, so the 60 percent of one study and the 53 percent of another are not commensurable. They also derive mainly from backend migrations, where the presentation layer is one concern among several rather than the object of the intervention.

2.3 Architectural Decisions in Global Environments

The third line addresses configuration at international scale. A review of fifteen primary studies filtered from 369 records credits modular monolithic architecture in cloud environments with simplified deployment and lower orchestration overhead, while noting the absence of consensus definitions and empirical benchmarks [23]. Hybrid practice combining domain-driven design, API management, containerization, orchestration, and observability receives comparable treatment [24], as does container-based transformation examined through healthcare and retail implementations [29]. Event-driven designs have been proposed for incremental decomposition [15].

Sector evidence converges. A review of 124 cloud ERP articles found that service-based architecture improves modularity and adaptability while addressing recognised shortcomings of monolithic designs [27]. Retail studies report lower deployment complexity where containerization is paired with CI/CD practice [28]. Multi-attribute fuzzy decision support ranked microservice architecture highest for electronic governance systems [29], and industry analysis of tooling and the Modulith alternative concurs [30].

2.4 Synthesis and Research Gap

Three observations follow. The direction of architectural travel is settled, and intermediate modular forms are now treated as legitimate destinations rather than incomplete migrations. Reported benefits cluster in a band of roughly a quarter to two-thirds on delivery-cycle measures, with engagement and reliability reported less systematically. Evaluation instruments exist but remain unstandardized.

The gap is narrower than the volume of literature suggests. Almost all of this work examines backend decomposition, where the unit of analysis is the service and the metric is latency, throughput, or deployment frequency. Studies treating the interface production pipeline as the primary object of architectural analysis are scarce, and none of those surveyed

does so at the scale of 180 national markets, where the design system rather than the service boundary is the coordinating mechanism. The present study addresses that setting.

3. METHODOLOGY

A change of architecture at the presentation layer alters how work moves through an organisation, and its effects appear in the production record before any runtime trace. The method adopted here, industrial case study system analysis, treats the designed system as the object of investigation and the operational record as evidence.

3.1 Research Design

The unit of analysis is the interface production pipeline of a single corporate platform serving 180 country sites in 42 languages, grouped into five regional clusters. A country site is called a market below, and a regional cluster a regional platform. The observation period spans five consecutive quarters. Two windows of equal length were defined within it: a baseline window covering the final two quarters of operation on the legacy stack, and a post-migration window covering the two quarters following completion of the cutover for the same set of markets. The intervening quarter, during which routes were transferred market by market, is reported in Figure 3 but excluded from the window comparison, since both stacks served production traffic throughout it. Equivalent windows on the same platform hold constant the product portfolio, the organisational structure, and seasonal demand, none of which a cross-company comparison could control.

Evidence came from four sources: version control and deployment records, publication logs of the content management system, the platform team inventory of interface components, and aggregated web analytics. No synthetic load testing was performed: where the object of change is the production pipeline, validated indicator sets distinguish process-level from runtime measures on precisely these grounds [5]. The baseline window contains 612 published localized pages and 8 production releases; the post-migration window contains 1,687 pages and 52 releases. Per-page indicators are computed over those page populations and per-release indicators over those release counts.

The procedure had three stages. The legacy architecture was decomposed into its coupling relations, so each delay could be traced to a structural cause rather than a scheduling accident. A target architecture was specified as a conceptual model, in the manner of frameworks that link method selection to the operating environment [3]. Indicators were then compared across the windows and interpreted against published ranges.

The cost structure of localized interface production motivates the design. Under the legacy stack every market required content preparation and a dedicated engineering implementation of the interface carrying it, so total cost grew as

$$C_L(n) = n(c_c + c_e) \tag{1}$$

where n is the number of served markets, cc is the content preparation cost for one market, and ce is the engineering cost of implementing the interface that presents it. Under a shared component library, the second term is paid once, and the same total becomes

$$C_T(n) = n\,c_c + c_e \tag{2}$$

with the same notation. The two expressions differ in one respect, and that difference is the substance of the intervention: expression (1) grows with the number of markets in both terms, expression (2) in one. What follows tests whether the platform behaves as expression (2) predicts.

3.2 Architectural Paradigm

The legacy configuration was a single WordPress installation in which templates, content, plugins, and delivery shared one deployable unit. Regional variation was handled through theme forks and per-market plugins. Each fork was a reasonable local decision and a collective liability: a correction to shared markup had to be applied in every fork, and the fork count grew with the market count.

The target architecture separates three responsibilities the monolith had combined. A Next.js rendering layer owns routing, page composition, and delivery, using static generation with incremental revalidation so that 180 locale trees are produced without per-request rendering cost. A headless content management system owns the content model and the editorial workflow, exposing locale-agnostic entities and their locale variants through a query API. A centralized design system, distributed as a versioned package, owns the interface vocabulary: tokens, primitives, composite components, and their accessibility behaviour.

The third element carries the coordination burden. In service-oriented migrations the bounded context keeps independently deployed units coherent, and stable modernization depends on drawing those boundaries along business lines [1]. At the presentation layer the design system performs the analogous function. A published component is a boundary: markets consume it rather than reimplement it, and a correction at the source reaches every market on the version in which it appears.

Migration was incremental rather than a replacement. Routes moved market by market behind a routing layer that dispatched traffic by path, the pattern modernization frameworks call the Strangler Fig [13] and refactoring work favours over disruptive rewrites [12]. Rollback stayed available for every market throughout.

Figure 1 places the three states side by side (fig. 1). In the legacy stack the template layer, the content database, and the plugin set share one deployment boundary from which 180 regional forks derive. In the transitional state a routing layer dispatches by path prefix, so migrated and unmigrated markets are served at once. In the target state the content repository, the design system package, and the rendering layer occupy separate boundaries.

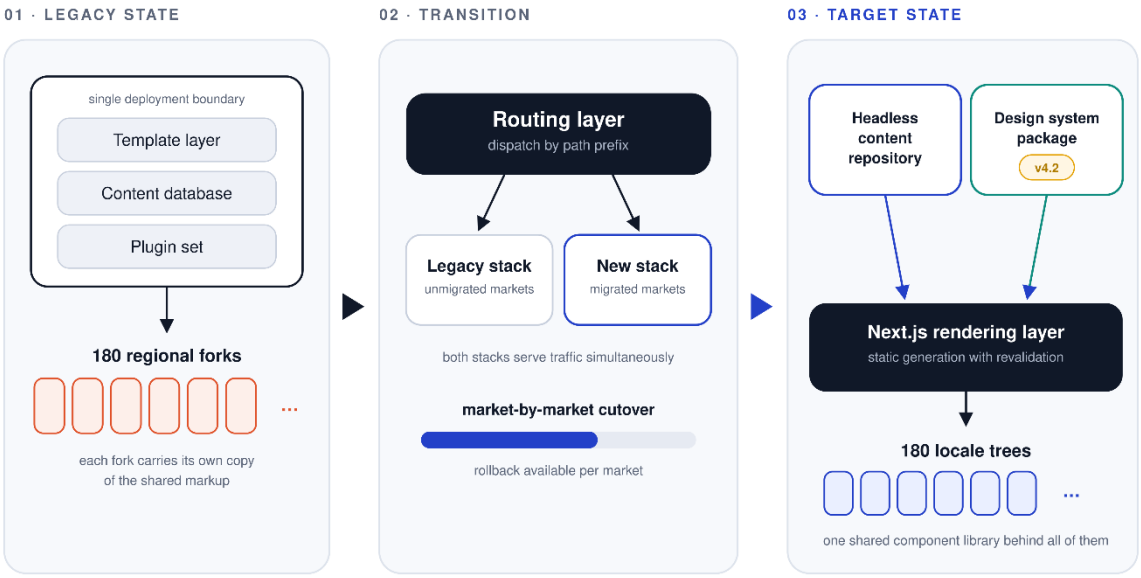

Fig 1: High-level flowchart of the migration architecture, from the monolithic WordPress stack to the decoupled Next.js and headless CMS delivery pipeline

3.3 Evaluation Criteria

Four criteria were selected, each tied to a structural property of the architecture rather than to a general notion of quality: delivery latency, the time a localized page takes to reach production; deployment autonomy, the share of that work proceeding without engineering involvement; component reuse, whether the design system acts as a boundary or only as a recommendation; and engagement, whether the new pipeline serves users at least as well as the old.

Delivery latency was decomposed rather than measured as a single quantity, since an aggregate conceals which stage causes a delay. For a given localized page,

$$T_{pub} = t_a + t_r + t_e + t_d \quad (3)$$

where Tpub is the elapsed working time from the opening of a content brief to publication, ta is authoring time for copywriting and translation, tr is editorial, legal, and regional approval time, te is the engineering time needed to implement or adapt the interface, and td is waiting time imposed by the release schedule. The architecture addresses the last two terms and leaves the first two untouched, which places an upper bound on any improvement it can produce.

Component reuse was quantified as the share of rendered interface instances originating in the shared library:

$$CRR = \frac{n_s}{n_t} \times 100\% \quad (4)$$

where CRR is the component reuse ratio, ns the number of rendered component instances resolved to the shared package, and nt the total number of rendered instances across all markets in the window. The measure comes from build manifests rather than source inspection, so components imported but never rendered do not inflate it.

Relative change between the two windows was computed uniformly for every indicator as

$$\Delta = \frac{|X_1 - X_0|}{X_0} \times 100\% \quad (5)$$

where X0 is the baseline value of an indicator, X1 its post-migration value, and Δ the magnitude of the change as a percentage of the baseline. Direction depends on the indicator: a decrease is favourable for latency and effort, an increase for cadence, reuse, and engagement. Window values are computed over all localized pages published within a window, so a window figure is not the arithmetic mean of the two quarterly medians in Figure 3.

4. RESULTS

4.1 Ecosystem Blueprint

The completed system resolves into three tiers with explicit contracts. The architecture is itself a finding, so it is described before the indicators. The content tier holds locale-agnostic entities, each with locale variants and a publication state, behind a single query interface. The composition tier, in Next.js, resolves a request path to a locale, retrieves the entities, and assembles a page from components rather than templates. The presentation tier is the design system package, versioned independently and consumed by every market through its regional platform.

Governance determines whether the arrangement holds: a shared library that any team may modify at will reverts to a fork under the pressure of 180 markets. The platform therefore adopted a two-level model, with a core set of primitives under central ownership and a regional extension layer in which markets may compose but not redefine. That division is an explicit ownership boundary of the kind whose absence reviews of modular practice identify as a recurring weakness [23], [24].

Figure 2 shows the ecosystem as a hub-and-spoke topology: the design system repository at the centre, design tokens, primitives, and composite patterns in concentric rings, a package registry whose spokes reach the regional platform clusters, and feedback arrows for patterns promoted from one market into the shared core (fig. 2). Each spoke carries its own version label, and the arrangement tolerates that skew because

the contract is the package interface, not a synchronised deployment.

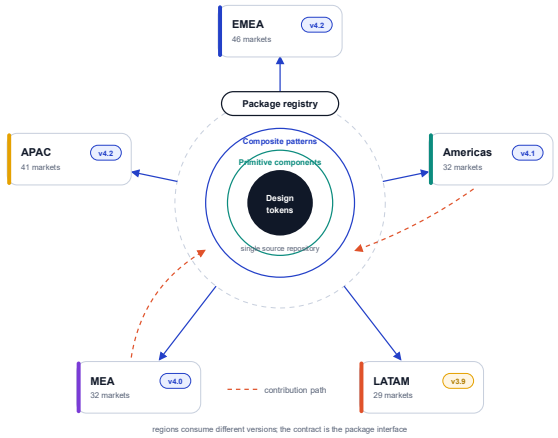

Fig 2: Topology of the centralized design system ecosystem and its distribution across regional platforms

4.2 Operational Efficiency Analysis

Table 1 reports the seven indicators across the baseline and post-migration windows, with relative change computed according to expression (5). Each per-page indicator aggregates 612 pages in the baseline window and 1,687 in the post-migration window.

Table 1. Comparative analysis of operational overhead and deployment times before and after structural modernization

Indicator	Legacy stack	Decoupled architecture	Absolute change	Relative change, %
Median time to publish a localized page, working days	20.0	15.0	-5.0	25
Engineering effort per localized page, person-hours	6.5	2.4	-4.1	63
Deployments to production per month	1.3	8.6	+7.3	562
Pages published without engineering involvement, %	11	78	+67	609
Component reuse ratio (CRR), %	34	77	+43	126
Regression defects reported per release	4.7	1.9	-2.8	60
Engaged sessions per visitor	2.10	2.94	+0.84	40

Decomposition according to expression (3) locates both the source of the reduction and its ceiling. Authoring time t_a and approval time t_r were unchanged at 11.5 working days combined, since translation and regional legal review are human processes untouched by the rendering stack. Engineering time t_e fell from 5.2 to 1.9 days and release waiting time t_d from 3.3 to 1.6. Those two stages account for 8.5 of the 20.0 baseline days, so the architecture could address 42.5 percent of elapsed time and removed 58.8 percent of that addressable portion. It performed close to its structural limit while the headline figure moved by 25 percent. The same arithmetic explains why the improvement is smaller than the 60 percent reported for composable enterprise architecture [16] or the 67 percent for a licensing release cycle [17]: those pipelines lack the human approval stages, which here consume 11.5 of the 15.0 days that remain, or 76.7 percent. Engineering days fell by 63.5 percent and engineering hours per page by 63.1 percent, which indicates that the saving came from reduced work content rather than from shorter queues in front of the same work.

The fall in engineering effort per page, the rise in cadence, and the growth in pages reaching production without engineering involvement together indicate that the release train ceased to be a queued shared resource. Two aggregate quantities make the point more directly. Published output rose from 612 to 1,687 localized pages, or 176 percent, while total engineering effort moved from 3,978 to 4,049 person-hours, a change of 1.8 percent, and pages delivered per market per window rose from 3.4 to 9.4. The architecture did not make engineers faster at a fixed workload. It removed the engineering term from the marginal page, which is what expressions (1) and (2) predict. Regression defects per release fell from 4.7 to 1.9 despite the higher cadence, consistent with smaller and more frequent deployments reducing the defect surface of each change [22], although normalising by output leaves a smaller gain, from 6.1 to 5.9 defects per hundred published pages.

Component reuse rose from 34 to 77 percent across the windows, computed as expression (4) over build manifests, and reached 82 percent in the final quarter. It is the mechanism behind the other indicators rather than an independent result, and the one most exposed to erosion: reuse measured soon after a migration reflects a library not yet pressed by market-specific demands, and sustaining it depends on the governance model rather than on any property of the tooling.

Figure 3 reports the same transition quarter by quarter. Panel (a) plots median publication time against the component reuse ratio: publication time falls from 20.4 and 19.6 working days in the baseline quarters to 18.5 during cutover and then to 15.4 and 14.6, while reuse rises from 33 and 35 percent to 52, 71, and 82. The series move in opposite directions, and the improvement is progressive after cutover rather than a step change at it, as regional teams shifted from adapting components to composing them. Panel (b) plots published pages against total engineering effort for the same quarters. Output rises from 298 to 891 pages while quarterly effort stays within 1,937 to 2,194 person-hours, peaking in the cutover quarter when migration and publication work overlapped. Effort per page therefore falls from 6.5 to 2.2 person-hours across the five quarters, and the two panels together indicate that the reuse ratio, rather than the rendering stack alone, carries the operational result (fig. 3).

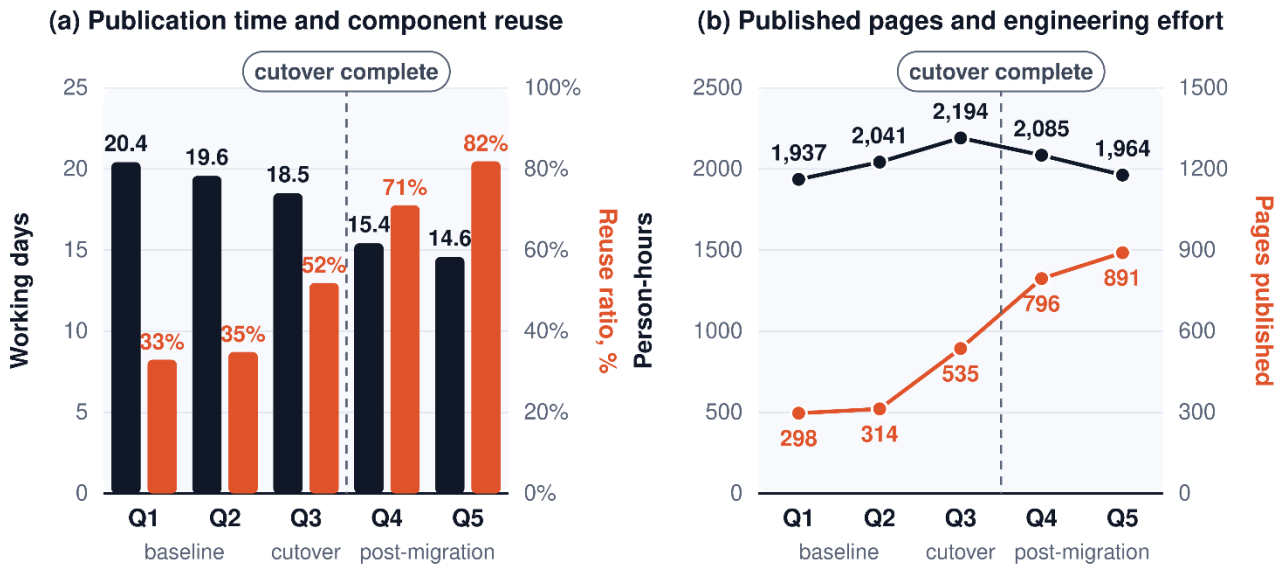

Fig 3: (a) Median time to publish a localized page and component reuse ratio by quarter; (b) localized pages published and total engineering effort by quarter. Q1 and Q2 form the baseline window, Q3 is the transitional quarter in which both stacks served production traffic, and Q4 and Q5 form the post-migration window. The dashed rule in each panel marks completion of the cutover

Attribution of the 40 percent rise in engaged sessions per visitor demands more caution. The migration coincided with faster delivery from static generation, with accessibility defects the shared library made possible to fix once, and with a navigation redesign the design system enabled but did not require. The rise is a joint outcome of architecture and the design decisions taken under it.

Table 2. Mapping of architectural changes to measured indicators and business outcomes

Architectural change	Indicator	Baseline to post	Post	Change	Business outcome
Headless content repository	Pages published without engineering, %	11	78	+67 pp	Publishing inside the campaign window
Design system as a versioned package	Component reuse ratio, %	34	77	+43 pp	One correction reaches all 180 markets
Per-market theme forks eliminated	Engineering effort per page, person-hours	6.5	2.4	-63 %	No engineering term per added market
Page composition from shared components	Localized pages published per window	612	1,687	+176 %	Output rose at 1.8 % more effort
Static generation with revalidation	Engaged sessions per visitor	2.10	2.94	+40 %	Engagement at flat infrastructure cost
Independent deployability per market	Deployments per month	1.3	8.6	6.6 times	Cadence no longer a queued resource
Release schedule decoupled from publication	Release waiting time, working days	3.3	1.6	-52 %	Pages ship inside the planning cycle
Two-level library governance	Defects per 100 published pages	6.1	5.9	-4 %	Quality held as output rose 2.76 times
Incremental cutover behind a routing layer	Markets cut over per week	0	13.8	180 in 13 weeks	Zero downtime, no delivery freeze

5. DISCUSSION

5.1 Alignment with Business Goals

An architectural decision earns its cost only if some business quantity moves. Table 2 traces each decision to the indicator it moves and to the outcome a non-technical stakeholder would recognise. The fifth row needs care. Defects per release fell by 60 percent, but published output rose by 176 percent over the same interval, so defect density fell only from 6.1 to 5.9 per hundred pages. Quality per unit of change improved sharply; quality per unit of delivered work held approximately constant. The two-level governance model in that row was not derived from an architectural principle. It was adopted after regional teams began requesting variants faster than the central team could absorb them, which is consistent with treating organisational alignment as a condition of technical success rather than a consequence of it [19].

5.2 Practical Implications

The findings correspond closely with composable commerce, where modular, API-first, cloud-native systems improved feature deployment speed by up to 53 percent alongside cost

savings [18]. Retail platforms and multi-market corporate platforms share a structural property: both present many variants of a small number of interface patterns, and both fail the same way when variation is handled by duplication rather than composition. Domain-driven, API-first architecture models report the same direction of effect [22], as do studies of containerized retail transformation paired with continuous delivery [28].

Three conditions appear necessary for the results to transfer. Interface variation across markets must be combinatorial rather than genuinely divergent; where markets need interfaces that differ in kind, a shared library becomes an obstacle instead of a boundary and a regionally partitioned arrangement may serve better [23]. Approval stages must be short enough for engineering stages to matter: expression (3) sets the ceiling, and an organisation whose legal review consumes thirty working days will not see a 25 percent improvement from any rendering stack. Ownership must be explicit, since a shared library without a governance model reverts to forks quietly enough to be detected only when reuse is measured.

The ordering carries a recommendation. Measure the composition of publication time before selecting an architecture, since that measurement determines whether the intervention addresses the binding constraint or a subordinate one. The instrument is inexpensive and the decision it informs is not. Reviews of modernization strategy place assessment before architectural selection for this reason [13], and decision-support methods formalise the same sequence when several candidates are in contention [26].

5.3 Limitations

This is a high-level architectural analysis, and its claims are bounded accordingly. The study observes one platform in one organisation across five quarters, four of which enter the window comparison. It has no control group, and it cannot separate the effect of the architecture from the effect of the attention the platform received during a funded modernization programme. Observational designs of this kind establish that an outcome followed a change; they do not establish that the change was sufficient to produce it. Engineering effort also held flat partly because migration work was charged to the same budget during the cutover quarter, so panel (b) of Figure 3 understates the capacity released once that work ended.

No code-level stress testing was performed. Runtime behaviour under traffic spikes, cold-start characteristics of incremental revalidation, and the resource profile of the rendering layer at peak remain unmeasured. Comparative studies show that such properties do not always improve with modularization and sometimes present quantified trade-offs [20], so the absence of measurement is not an absence of cost. Stress testing of the production system is the natural extension, and validated indicator sets exist to structure it [5]. Two indicators carry specific caveats. Engagement is a joint outcome of the architecture and the design decisions taken under it, and the analysis cannot apportion the 40 percent increase between them. Component reuse was measured shortly after cutover, before years of market-specific requests had pressed the library, and reuse ratios decay under sustained pressure. A measurement across eight or twelve quarters would establish whether the governance model holds; this study cannot.

Because the field lacks standardized metrics, datasets, and baselines [2], [23], every case study reports against a local baseline and cross-study comparison remains approximate.

6. CONCLUSIONS

When content production, interface implementation, and delivery are bound in one deployable unit, the cost of serving an additional market includes a full engineering implementation, and the queue of pending localized pages lengthens with every market added. This study evaluated a structural migration from that configuration to a decoupled architecture combining a Next.js rendering layer, a headless content repository, and a centralized design system distributed as a versioned package, assessed through system analysis of the production pipeline.

Median time to publish a localized page fell by 25 percent, against the 42.5 percent of elapsed time the architecture was able to address at all, while published output rose by 176 percent at an engineering budget that changed by 1.8 percent. Table 1 reports the remaining indicators. The transferable finding is narrower than the indicator set: under the legacy configuration the marginal cost of an additional market carried a full engineering term, as expression (1) states, and under a shared component library it does not, as expression (2) states. This holds where interface variation is combinatorial rather than divergent, where approval stages leave room for engineering time to matter, and where the shared library is governed as an owned boundary. Further work should test the delivered system under production load and track component reuse across a longer horizon.

7. REFERENCES

[1] Guntakandla, A. R. 2025. Modular architecture: A scalable and efficient system design approach for enterprise applications. World Journal of Advanced Research and Reviews. DOI: 10.30574/wjarr.2025.26.1.1340.

[2] Jayaprakash, D. B. 2025. Best practices for legacy system modernization in enterprise IT. European Modern Studies Journal. DOI: 10.59573/emsj.9(4).2025.84.

[3] Abduaziz, A. 2026. Strategies for migrating monolithic enterprise applications to modular microservices architecture. International Journal of Computer Science & Information System. DOI: 10.55640/ijcsis/Volume1Issue05-02.

[4] Puchakayala, S. S. 2026. Review of modern strategies for migrating monolithic enterprise applications to microservices architecture: Challenges, patterns, and best practices. International Journal of Advanced Engineering Management and Science. DOI: 10.22161/ijaems.123.5.

[5] Bjørndal, N., de Araújo, L. J. P., Bucchiarone, A., Dragoni, N., Mazzara, M., and Dustdar, S. 2021. Benchmarks and performance metrics for assessing the migration to microservice-based architectures. Journal of Object Technology. DOI: 10.5381/jot.

[6] Ayyarrappan, M. A. 2020. Transitioning from monolithic to microservice architectures. Journal of Software Engineering and Simulation. DOI: 10.35629/3795-06034952.

[7] Abgaz, Y. M., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G. M., Yilmaz, M., Buckley, J., and Clarke, P. M. 2023. Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering. DOI: 10.1109/TSE.2023.3287297.

[8] Kaloudis, M. 2024. Evolving software architectures from

monolithic systems to resilient microservices: Best practices, challenges and future trends. International Journal of Advanced Computer Science and Applications. DOI: 10.14569/IJACSA.2024.0150901.

[9] Hassan, H., Abdel-Fattah, M. A., and Mohamed, W. 2024. Migrating from monolithic to microservice architectures: A systematic literature review. International Journal of Advanced Computer Science and Applications. DOI: 10.14569/IJACSA.2024.0151013.

[10] Agaev, A. 2026. Strategies for migrating monolithic software systems to a microservice architecture. International Journal of Advanced Engineering, Management and Science, 12, 1, 85-92. DOI: 10.22161/ijaems.121.11.

[11] Faustino, D., Gonçalves, N., Portela, M., and Silva, A. R. 2024. Stepwise migration of a monolith to a microservices architecture: Performance and migration effort evaluation. Performance Evaluation, 164, 102411. DOI: 10.1016/j.peva.2024.102411.

[12] Cakar, C. 2025. From monolith to modular infrastructure: Refactoring strategies for scalable and secure software evolution. Iconic Research and Engineering Journals, 8, 10. DOI: 10.64388/IREJ8110-1715576.

[13] Chary, A., and Suroju, A. C. 2025. Legacy application modernization: A strategic framework for enterprise transformation. World Journal of Advanced Research and Reviews. DOI: 10.30574/wjarr.2025.26.2.2042.

[14] Hassan, H., Abdel-Fattah, M. A., and Mohamed, W. 2025. A pattern-based framework for automated migration of monolithic applications to microservices. Big Data and Cognitive Computing. DOI: 10.3390/bdcc9100253.

[15] Christian, J., Steven, Kurniawan, A., and Anggreainy, M. S. 2023. Analyzing microservices and monolithic systems: Key factors in architecture, development, and operations. In Proceedings of the 6th International Conference of Computer and Informatics Engineering (IC2IE). DOI: 10.1109/IC2IE60547.2023.10331155.

[16] Wang, W.-J. 2025. Achieving business scalability with composable enterprise architecture. IEEE Access. DOI: 10.1109/ACCESS.2025.3564710.

[17] Lakshmi, L., and Mangalagiri, C. K. 2025. Scaling enterprise licensing: Modernizing volume licensing platforms (2010-2015). Journal of Information Systems Engineering & Management. DOI: 10.52783/jisem.v10i62s.13708.

[18] Bathina, S. 2025. Composable commerce architectures: Building agile retail systems. Journal of Information Systems Engineering & Management. DOI: 10.52783/jisem.v10i57s.12317.

[19] Bhatt, S. 2026. From monolithic middleware to cloud-native microservices: A performance-driven modernization study. In Proceedings of the International Symposium on Digital Forensics and Security. DOI: 10.1109/ISDFS69419.2026.11459056.

[20] Tapia, F., Mora, M., Fuertes, W., Aules, H., Flores, E. S., and Toulkeridis, T. 2020. From monolithic systems to microservices: A comparative study of performance. Applied Sciences. DOI: 10.3390/app10175797.

[21] Slamaa, A. A., El-Ghareeb, H. A., and Saleh, A. A. 2021. A roadmap for migration system-architecture decision by neutrosophic-ANP and benchmark for enterprise resource planning systems. IEEE Access. DOI: 10.1109/ACCESS.2021.3068837.

[22] Babatunde, O., Olamijuwon, J., Cadet, E., Osundare, O. S., and Ekpobimi, H. O. 2024. Building a microservices architecture model for enhanced software delivery, business continuity and operational efficiency. International Journal of Frontiers in Engineering and Technology Research. DOI: 10.53294/ijfetr.2024.7.2.0050.

[23] Al-Qora'n, L. F., and Al-Said Ahmad, A. 2025. Modular monolith architecture in cloud environments: A systematic literature review. Future Internet, 17, 11, 496. DOI: 10.3390/fi17110496.

[24] Beeraka, B. S. 2025. Modernizing enterprise architecture: A guide to microservices migration and hybrid cloud integration. International Journal of Science and Research Archive. DOI: 10.30574/ijsra.2025.14.1.0052.

[25] Vutti, V. R. 2024. Enterprise application modernization: A journey through container-based cloud architecture transformation. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 10, 6, 1666-1674. DOI: 10.32628/CSEIT241061209.

[26] Ghosh, A. 2025. Event-driven architectures for microservices: A framework for scalable and resilient rearchitecting of monolithic systems. International Journal for Sciences and Technology. DOI: 10.71097/IJSAT.v16.11.2498.

[27] Lee, C., Kim, H. F., and Lee, B. 2024. A systematic literature review on the strategic shift to cloud ERP: Leveraging microservice architecture and MSPs for resilience and agility. Electronics. DOI: 10.3390/electronics13142885.

[28] Veeramreddygari, U. 2025. Retail digital transformation through microservices and containerized architectures. In Proceedings of the 6th International Conference on Data Intelligence and Cognitive Informatics (ICDICI). DOI: 10.1109/ICDICI66477.2025.11135409.

[29] Tyagi, N. K., and Tyagi, K. 2024. A systematic multi attributes fuzzy-based decision-making to migrate the monolithic paradigm electronic governance applications to new software architecture. Concurrency and Computation: Practice and Experience. DOI: 10.1002/cpe.8294.

[30] Ulichev, O., Dorenskyi, O., and Kulahin, V. 2024. Innovative solutions and benefits of microservice architecture for software products. Central Ukrainian Scientific Bulletin. Technical Sciences, 10(41).1, 16-29. DOI: 10.32515/2664-262X.2024.10(41).1.16-29.