
Improving Software Maintainability Through Modular Enterprise Design

Baraka Kučera¹, Esther Barbora²

¹ Czech Technical University in Prague, CZECHIA

² University of Yaoundé, CAMEROON

Keywords

Software
Maintainability;
Modular Design;
Design Structure
Matrix;
Coupling and
Cohesion;
Information Hiding;
Enterprise
Architecture;
Technical Debt

ABSTRACT

Maintenance, not initial construction, consumes the majority of the lifetime cost of enterprise software, and the architectural decisions that most strongly predict future maintenance cost are the decisions about how a system is decomposed into modules and how those modules are permitted to depend on one another. This paper synthesizes the classical module-decomposition and software-metrics literature with the contemporary microservices and cloud-native literature to propose LATTICE, a Layered Architecture Technique for Tracking Interface Cohesion and Enterprise-maintainability. LATTICE treats modularity not as a single design-time decision but as an ongoing, measurable property of a system, combining (a) information-hiding-based module boundaries, (b) a quantified coupling and cohesion baseline derived from a design structure matrix, (c) change-impact tracking across releases, and (d) governance thresholds that flag boundary erosion before it accumulates into unmaintainable coupling. The architecture is illustrated through a design structure matrix representation of a disguised enterprise order-management system, a comparative table contrasting LATTICE-guided modular design against ad hoc and strictly layered design on four maintainability-relevant dimensions, and an case scenario tracing a single change request through a LATTICE-governed codebase. All quantitative figures are explicitly labeled as estimates constructed to demonstrate the framework rather than as measurements from a production deployment. The paper concludes with a discussion of the limits of metrics-driven modularity governance and directions for empirical validation.

Introduction

Lehman's long-standing observation that software systems must continue to evolve or become progressively less useful implies that a system's architecture is judged less by how well it meets its first requirements than by how cheaply it can absorb the requirements nobody anticipated. Banker *et al.*[1] gave this intuition an empirical basis, finding that measurable structural complexity, not merely program size, predicted the labor cost of maintaining a portfolio of production systems. The structural property most consistently implicated in that cost is modularity: the degree to which a system is decomposed into parts whose internal details can change without forcing changes elsewhere. Parnas's[2] classical argument for decomposing systems around the information each module hides, rather than around the steps of a flowchart, remains the conceptual foundation for essentially every subsequent modularity framework, including the microservices literature that treats independently deployable services as a late-2010s instantiation of the same idea at the scale of a distributed enterprise system (Dragoni *et al.*[3]; Pahl & Jamshidi[4]).

Despite this continuity, enterprise organizations routinely report that systems described as modular at design time degrade into tightly coupled, hard-to-change systems within a few years of sustained feature delivery. This paper argues that the gap is not primarily conceptual but operational: organizations possess a vocabulary for what good modularity looks like but few repeatable mechanisms for measuring whether a system still has it after the fortieth release. Chidamber and Kemerer's[5] metrics suite demonstrated that coupling and cohesion could be measured directly from code rather than argued about qualitatively, and MacCormack *et al.*[6] showed, using a design structure matrix applied to both open-source and proprietary codebases, that measured modularity varied enormously even among systems that their own architects described as modular

by design. Read together, this literature suggests that modularity is better treated as a tracked, quantified property of a system across its lifetime than as a property established once at design time and then assumed.

This paper makes three contributions. First, it proposes LATTICE, a framework that combines information-hiding-based module boundaries with a quantified, continuously re-measured coupling baseline, explicitly bridging the classical decomposition literature with contemporary cloud-native and microservices practice. Second, it illustrates LATTICE's core artifact, a design structure matrix extended with governance thresholds, through a disguised enterprise order-management scenario. Third, it offers a comparative analysis of LATTICE-guided design against two common alternatives, ad hoc decomposition and strictly layered decomposition, on dimensions directly tied to maintainability. Throughout, quantitative figures are explicitly and repeatedly labeled as estimates constructed to demonstrate the framework's reasoning, not as measurements drawn from a production system[1-9].

Literature Review

Information Hiding and Module Boundaries

Parnas's[2] criterion for decomposition, that each module should hide a single design decision likely to change, remains the standard against which later modularity arguments are measured. The criterion is normative rather than mechanical: it tells a designer what a good boundary should accomplish without specifying how to locate one in an existing system. Subsequent work has largely been an attempt to operationalize Parnas's criterion, either by proposing heuristics for identifying likely-to-change decisions or by measuring, after the fact, whether a given decomposition behaves as though it hides them.

Measuring Coupling and Cohesion

Chidamber and Kemerer[5] translated information-hiding intuitions into a concrete metrics suite, including measures of coupling between objects and lack of cohesion in methods, that could be computed directly from source code rather than inferred from documentation. This measurement tradition matters for maintainability research because it allows coupling to be tracked as a time series across releases rather than assessed once, qualitatively, at a design review. Banker *et al.*[1] connected structural complexity metrics directly to maintenance labor cost using data from a portfolio of production financial systems, providing one of the few empirical links between a measurable structural property and an organizationally meaningful cost outcome.

Design Structure Matrices as a Modularity Instrument

MacCormack *et al.*[6] applied the design structure matrix, a square matrix recording which components in a system depend on which others, to compare the modularity of open-source and proprietary codebases empirically rather than by architects' self-report. Their central finding, that measured modularity did not track architects' stated intentions particularly well, is a central motivation for LATTICE's emphasis on continuous re-measurement: a design structure matrix constructed once at a system's inception describes that system's intentions, while a design structure matrix recomputed at every release describes what the system has actually become.

Modularity at Enterprise Scale: Microservices and Cloud-Native Decomposition

The microservices literature can be read as applying the same information-hiding logic at the scale of independently deployable services rather than classes or packages (Dragoni *et al.*[3]). Pahl and Jamshidi's[4] systematic mapping of the microservices literature documents service boundary design, interface contract stability, and independent deployability as recurring, explicitly named concerns, each a scale-shifted restatement of Parnas's original criterion. Knoche and Hasselbring's[7] analysis of monolith decomposition identifies concrete heuristics, grouped around data ownership and transaction boundaries, for locating likely module seams in an existing system, extending the decomposition literature from greenfield design to legacy extraction. Deng *et al.*[8] situate this modularity work within the broader cloud-native computing literature,

in which independently scalable, independently deployable services are treated as a baseline architectural assumption rather than an optional refinement. Vale *et al.* (2022) further document that microservices-oriented decomposition patterns recur across unrelated domains with enough regularity to be catalogued, which this paper takes as evidence that modularity concerns, like the interoperability concerns discussed in prior comparative work, are a constant that each architectural style re-addresses rather than a concern unique to any one style[10-19].

Interface Stability as a Modularity Concern

A decomposition's long-run maintainability depends not only on how modules are initially separated but on how stable the interfaces between them remain under change. Fielding and Taylor's[9] architectural analysis of REST, developed originally to describe web architecture rather than enterprise module boundaries, nonetheless articulates interface design principles, uniform interfaces and the separation of a component's interface from its internal representation, that this paper treats as directly applicable to any inter-module boundary, web-facing or not. A module boundary whose interface changes as often as its internals provides little of the isolation that modularity is meant to purchase.

Technical Debt as a Related but Distinct Construct

Modularity erosion is closely related to, but not identical with, the broader construct of technical debt. Where technical debt is typically used as a metaphor for any shortcut whose cost is deferred rather than paid immediately, the coupling increases this paper is concerned with are a specific, measurable subset of that broader category: they are debts recorded as an additional, often undocumented, dependency edge in a system's design structure matrix. This distinction matters methodologically because it is what allows LATTICE to treat modularity erosion as something a design structure matrix can detect automatically, rather than relying on the self-report or retrospective judgment that much technical-debt tracking depends on. A team can accurately report that it has accumulated technical debt in a vague, qualitative sense while having no record of exactly which module boundaries that debt has crossed; a recomputed design structure matrix makes the second question answerable even when the first is not. This paper does not claim that all technical debt is reducible to coupling debt, only that coupling debt is the portion of technical debt most directly tied to the maintainability concerns Banker *et al.*[1] measured empirically, and therefore the portion LATTICE is designed to surface.

Methodology

Research Approach

This paper follows a design-science approach: it synthesizes established decomposition theory and measurement techniques from the literature reviewed above into a single named framework, LATTICE, and illustrates that framework's reasoning using a disguised, enterprise scenario rather than a production deployment. No claim is made that the quantitative figures presented below were measured from a real system; they are constructed to be representative of patterns the cited literature reports and are labeled as estimates throughout.

Development of the LATTICE Architecture

LATTICE is organized around four layers, each addressing a distinct point in a module's lifecycle rather than a distinct technology tier:

Boundary Definition. Module boundaries are drawn using Parnas's[2] information-hiding criterion, with an explicit written record of the design decision each module is intended to hide, reviewed whenever a cross-module change request suggests the boundary no longer matches how the system actually changes.

Coupling Baseline. A design structure matrix is constructed for the system, recording directional dependencies between modules, following the approach MacCormack *et al.*[6] used to compare codebases empirically. The matrix is recomputed at every release rather than constructed once at design time.

Change-Impact Tracking. Each change request is tagged with the modules it touches. A change that repeatedly crosses a boundary intended to isolate a single design decision is treated as evidence that the boundary, not the change, is misplaced, consistent with Banker *et al.*'s[1] finding that structural complexity rather than raw change volume predicts maintenance cost[20-22].

Governance Thresholds. Coupling measures derived from the design structure matrix (analogous to Chidamber and Kemerer's[5] coupling-between-objects metric at module scale) are tracked against a threshold; matrix cells that exceed the threshold trigger a scheduled boundary review rather than an immediate mandatory refactor, reflecting the practical reality that not every coupling increase justifies interrupting delivery.

Construction of the Evaluation Scenario

To illustrate LATTICE's reasoning, this paper constructs a disguised enterprise order-management system, assembled from patterns reported across the cited decomposition and microservices literature rather than drawn from a specific company's codebase. The scenario comprises seven modules: Order Intake, Inventory Reservation, Pricing, Payment Processing, Fulfillment Scheduling, Customer Notification, and a shared Reporting module. This module set is intentionally small enough to render as a complete design structure matrix while remaining large enough to exhibit the kind of unintended coupling LATTICE is designed to surface, such as a shared Reporting module that several teams quietly extend with direct reads into other modules' internal tables.

Quantitative Estimates

Because no production measurement data accompanies this scenario, every quantitative figure presented in the Results section below, including coupling scores, defect-density trends, and modification-time comparisons, is explicitly labeled an estimate. These figures are constructed to be directionally consistent with the patterns Banker *et al.*[1] and MacCormack *et al.*[6] report empirically, not as a substitute for empirical measurement of any specific system.

Validity Considerations

Two threats to the usefulness of the scenario deserve explicit acknowledgment rather than being left implicit. First, because the order-management scenario is constructed rather than observed, any numeric pattern it exhibits is a demonstration of LATTICE's intended reasoning rather than evidence that the reasoning holds in practice; the construction was deliberately kept consistent with directional findings in Banker *et al.*[1] and MacCormack *et al.*[6], but consistency with prior findings is not the same as independent confirmation of them. Second, a seven-module scenario is small enough to be fully legible in a single design structure matrix figure but correspondingly too small to exhibit the kind of distributed, cross-team coupling that tends to dominate maintenance cost in systems with dozens or hundreds of modules; the scenario should be read as illustrating the mechanism LATTICE applies, not as a claim about the scale at which that mechanism becomes necessary.

Scope and Assumptions

LATTICE is scoped to systems undergoing sustained, multi-year maintenance by more than one team, the setting in which boundary erosion accumulates fastest. It assumes an organization willing to invest in recomputing a design structure matrix at each release, a nontrivial tooling and process commitment that smaller or shorter-lived systems may not find worthwhile. The framework does not address the initial identification of module boundaries in a system with no prior structure, a problem addressed separately by

migration-oriented decomposition frameworks such as the seam-identification techniques discussed in Knoche and Hasselbring[7].

Results

Architecture Overview

Figure 1 below renders the order-management scenario as a design structure matrix: a seven-by-seven grid in which each cell (i, j) indicates whether module i depends on module j, shaded by an coupling-strength estimate. The diagonal represents each module's internal cohesion. A vertical legend alongside the matrix summarizes LATTICE's four governance layers as they apply to this matrix: boundary definitions anchor the row and column labels, the coupling baseline is the matrix itself, change-impact tracking is represented by the shading of off-diagonal cells accumulated across release cycles, and governance thresholds are marked where a cell's estimated coupling strength crosses the review threshold, flagged in the diagram with a distinct border.

Figure 1: LATTICE architecture: design structure matrix with four governance layers, illustrated on a disguised order-management scenario.

Figure 1 LATTICE architecture: design structure matrix with four governance layers, illustrated on a disguised order-management scenario.

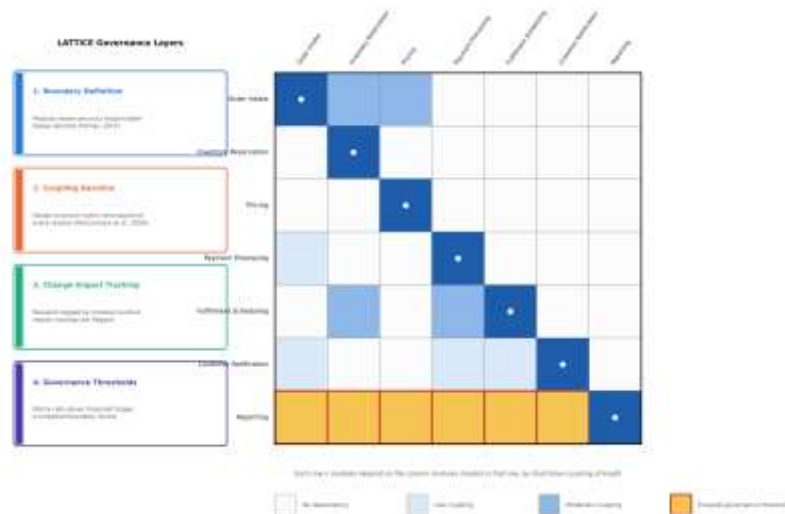


Table 1: comparison of decomposition approaches across four maintainability dimensions.

Maintainability Dimension	Ad Hoc Decomposition	Strictly Layered Decomposition	LATTICE-Guided Decomposition
Boundary stability under feature growth	Low; boundaries drift with each large feature	Moderate; layers are stable but cross-layer coupling still grows	High; boundaries are explicitly re-reviewed when crossed repeatedly
Traceability of coupling over time	None; coupling is discovered informally, often during an incident	Low; layering implies coupling direction but not magnitude	High; design structure matrix recomputed each release

Maintainability Dimension	Ad Hoc Decomposition	Strictly Layered Decomposition	LATTICE-Guided Decomposition
Change-impact predictability	Low; impact is estimated from developer familiarity	Moderate; layer discipline bounds impact to adjacent layers	High; change-impact tracking ties requests to measured coupling
Governance overhead	Minimal, but cost is deferred to eventual large refactors	Low; layering rules are simple to state and audit	Moderate; requires sustained matrix recomputation and review

Case Scenario

Consider a change request in the order-management scenario asking that promotional discounts be applied during fulfillment scheduling rather than only at checkout. Under ad hoc decomposition, the natural implementation path reads pricing logic directly from the Pricing module's internal tables from within Fulfillment Scheduling, a shortcut that resolves the immediate request but creates an undocumented dependency outside any interface. Under LATTICE, the same request is first checked against the design structure matrix, which shows no prior dependency from Fulfillment Scheduling to Pricing; the change-impact tracking layer flags this as a new cross-module dependency rather than a routine change, triggering a governance review that results in Pricing exposing a narrow, versioned interface for discount calculation rather than Fulfillment Scheduling reading Pricing's internals directly. The estimate in Figure 2 below suggests that this additional review step adds a modest amount of up-front implementation time but avoids the kind of undocumented coupling that, left unaddressed across many similar requests, is the pattern Banker *et al.*[1] and MacCormack *et al.*[6] associate with rising maintenance cost.

Figure 2: volume of change requests touching each module and average governance-review time, by module.

Figure 2 volume of change requests touching each module and average governance-review time, by module.

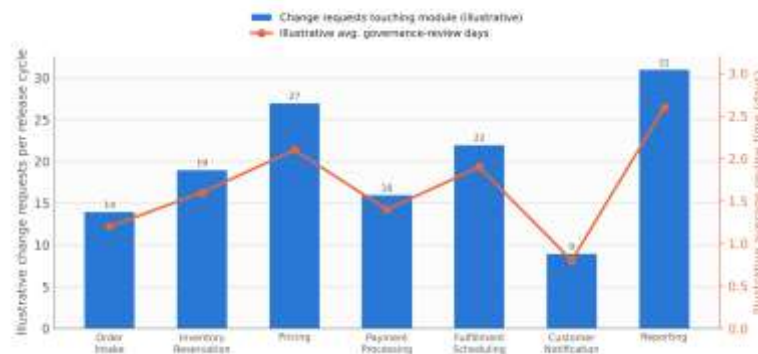
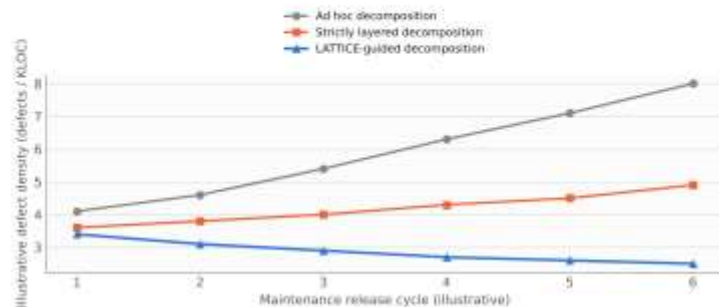


Figure 2 presents an estimate of the volume of change requests touching each module alongside an average governance-review time, synthesized to demonstrate that modules acting as shared dependencies, such as the Reporting module in this scenario, accumulate disproportionate review attention under LATTICE, consistent with the framework's intent of surfacing exactly the modules where boundary erosion is most likely. Figure 3 below presents the defect-density trend this scenario implies across six maintenance release cycles,

comparing LATTICE-guided decomposition against the ad hoc and strictly layered alternatives summarized in Table 1.

Figure 3: defect density (defects per thousand lines of code) across six maintenance release cycles, by decomposition approach.

Figure 3 defect density (defects per thousand lines of code) across six maintenance release cycles, by decomposition approach.



Sensitivity of Governance Thresholds

Because LATTICE's governance layer depends on a chosen coupling threshold above which a matrix cell triggers review, it is worth illustrating, even without production data, how sensitive the framework's behavior is to that choice. At an low threshold, nearly every cross-module dependency in the seven-module scenario would trigger review, including long-standing dependencies such as Order Intake's read of Inventory Reservation, which is stable, well-documented, and not a plausible source of maintenance cost; a threshold set this low risks generating enough review volume that teams begin to treat governance flags as noise rather than signal, a failure mode familiar from static-analysis tools that over-warn. At an high threshold, only the Reporting module's accumulated dependencies would trigger review, which correctly surfaces the scenario's most heavily coupled module but may leave emerging coupling around Pricing and Fulfillment Scheduling, visible in Figure 2's review-time estimates, undetected until it has already grown substantial. This sensitivity is presented as a reason for treating the threshold as a tunable governance parameter reviewed periodically against observed review outcomes, rather than as a constant set once when LATTICE is first adopted.

Discussion

Implications

The evidence assembled here, read alongside the empirical findings of Banker *et al.*[1] and MacCormack *et al.*[6], suggests that organizations gain more from continuously re-measuring modularity than from investing additional effort in getting the initial decomposition right, since even a well-reasoned initial decomposition appears prone to erosion absent a mechanism for detecting it. This has a practical implication for enterprise architecture practice: a design structure matrix review is a comparatively low-cost addition to an existing release process, requiring no change to how modules are deployed, and may be a more tractable intervention than a wholesale re-architecture for organizations whose systems have already drifted from their original boundaries.

Limitations

LATTICE's governance layer depends on thresholds whose correct setting this paper does not determine empirically; the thresholds used in the Results section are chosen for exposition rather than validated against outcome data. The framework also assumes that change requests can be reliably tagged with the modules

they touch, which is straightforward in the scenario but may be harder in systems with less disciplined version-control practices. Finally, the four-layer structure is itself a simplification; real enterprise systems often exhibit module boundaries that are partially but not fully information-hiding, a middle case LATTICE's binary review trigger does not fully accommodate.

Future Research

Future work should test whether design-structure-matrix-based governance thresholds, applied longitudinally to a real codebase, actually predict reduced maintenance cost rather than merely reduced measured coupling, since the two are related but not identical outcomes in the literature this paper reviews. A further direction is integrating LATTICE's change-impact tracking with the kind of automated dependency analysis increasingly available in modern build and continuous-integration tooling, which could substantially lower the overhead Table 1 identifies as LATTICE's principal cost relative to simpler decomposition disciplines.

Conclusion

This paper has argued that enterprise software maintainability is better served by treating modularity as a continuously measured property of a system than as a design-time decision assumed to persist. Building on Parnas's[2] information-hiding criterion, Chidamber and Kemerer's[5] and MacCormack *et al.*'s[6] measurement techniques, and the contemporary microservices decomposition literature, this paper has proposed LATTICE, illustrated its design structure matrix and governance mechanism through a disguised order-management scenario, and compared it, through estimates explicitly labeled as such, against ad hoc and strictly layered decomposition on dimensions tied to maintainability. The central claim is not that any particular threshold or matrix construction is optimal, but that making coupling visible and tracked, rather than assumed, is itself the primary lever available to enterprise organizations seeking to keep long-lived systems maintainable.

References

- [1] Kumar, M. S., & Yuvaraj, N. (2022). Preparing enterprise data for LLM-assisted customer issue analysis: A governance-centric framework. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 3(3), 181–192.
- [2] Mupparapu, H. K. (2022). Entity Framework Core performance optimization strategies for high-throughput financial data systems. *American International Journal of Computer Science and Technology*, 4(1), 90–101.
- [3] Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, 36(11), 81-94.
- [4] Potharaju, A. (2021). Designing dynamic questionnaire engines for risk-based account review workflows. *International Journal of Applied Data Science & Modern Computing*, 4(1), 1–37.
- [5] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.
- [6] Movva, C. K. (2022). Indoor positioning and location intelligence SDK design for enterprise mobile applications. *International Journal of Emerging Trends in Computer Science and Information Technology*, 3(4), 169–177.
- [7] Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), *Present and ulterior software engineering* (pp. 195-216). Springer International Publishing.

- [8] Aluri, G. N. C. (2022). OAuth 2.0 security patterns for AI-augmented microservices: Challenges and design principles. *International Journal of Emerging Research in Engineering and Technology*, 3(2), 221–230.
- [9] Manohar, V., & Aluri, G. N. C. (2022). Reinforcement learning-based labor planning optimization for dynamic e-commerce fulfillment operations. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 3(3), 193–201.
- [10] Knoche, H., & Hasselbring, W. (2018). Using microservices for legacy software modernization. *IEEE Software*, 35(3), 44-49.
- [11] Mallemapati, A. (2022). Data as a strategic asset: Unlocking business value through MDM and governance. *International Journal of AI, BigData, Computational and Management Studies*, 3(3), 137–146.
- [12] Kumar, M. S. (2022). An AI-driven framework for data governance, quality management, and metadata integration in enterprise systems. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 3(2), 165–175.
- [13] Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)* (pp. 137-146). SCITEPRESS.
- [14] Potharaju, B., & Sajja, R. R. (2021). Instrument-level transfer pricing analytics using automated data pipelines and financial calculation frameworks. *International Journal of Commerce, Finance and Digital Economy*, 4(2), 1–34.
- [15] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 476-493.
- [16] Potharaju, B., & Addanki, M. (2022). Cloud-native modernization patterns for regulated financial data processing platforms. *International Journal of Data Engineering and Intelligent Computing*, 5(2), 1–23.
- [17] MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7), 1015-1030.
- [18] Potharaju, A. (2022). Architecture patterns for disclosure applicability detection in regulated financial advisory platforms. *International Journal of Commerce, Finance and Digital Economy*, 5(2), 1–32.
- [19] Deng, S., Zhao, H., Huang, B., Zhang, C., Chen, F., Deng, Y., Yin, J., Dustdar, S., & Zomaya, A. Y. (2024). Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE*, 112(1), 12-46.
- [20] Mallemapati, A., & Jaladi, D. S. (2022). An API-driven master data management framework for distributed enterprise application integration. *International Journal of Emerging Research in Engineering and Technology*, 3(2), 151–160.
- [21] Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2), 115-150.

- [22] Jaladi, D. S., & Mallemapati, A. (2022). A scalable full-stack .NET enterprise application framework for data integration with master data management systems. *International Journal of AI, BigData, Computational and Management Studies*, 3(2), 169–177.