
AI-Driven Enterprise Application Modernization: Integrating Cloud-Native Microservices, Intelligent Data Pipelines, and Secure Infrastructure

Anna Jacobs¹, Gruber Moser^{2*}

¹ TU Wien, AUSTRIA

² KU Leuven, BELGIUM

ABSTRACT

Large enterprises continue to operate substantial estates of monolithic, tightly coupled applications even as cloud-native microservices, intelligent data pipelines, and zero-trust security models have become the accepted architectural baseline for new systems. Modernization efforts that address these three concerns separately — decomposing a monolith into services, rebuilding its data flows, and retrofitting its security posture — tend to produce a system that is cloud-native in form but still fragile in practice, because the boundaries chosen for the service decomposition rarely align with the boundaries a well-designed data pipeline or a zero-trust access model would require. This paper proposes TRIMOD (Tri-Layer Intelligent Modernization), an architecture in which an AI-assisted decomposition layer, an intelligent data-pipeline layer, and a zero-trust secure-infrastructure layer are designed jointly rather than sequentially, coordinated by a central modernization orchestration engine that uses the same service-boundary analysis to inform microservice decomposition, pipeline topology, and access-policy scoping at once. Following a design-science research approach, the architecture is illustrated, not empirically validated, through a comparative capability table, two illustrative figures, and a short case scenario, with every quantitative value explicitly labeled as an illustrative estimate grounded in the cited literature's directional findings rather than as measured data from a production system. The paper situates TRIMOD against the literature on microservice decomposition, AI-assisted software modernization, intelligent data-pipeline architecture, and zero-trust infrastructure, and discusses the framework's practical implications, assumptions, and limitations.

Keywords: Enterprise Modernization; Microservices; Cloud-Native Architecture; Data Pipelines; Zero Trust; Large Language Models; Software Architecture

Introduction

Most large enterprises did not start with cloud-native architecture; they arrived at it, gradually, from a position of substantial existing investment in monolithic or loosely modernized applications that continue to run core business processes. Replacing these systems outright is rarely feasible, so modernization typically proceeds incrementally: a monolith is decomposed into services, its data flows are rebuilt around modern pipeline technology, and its security model is retrofitted toward the zero-trust principle that no request should be implicitly trusted regardless of its network origin[1]. Each of these three efforts has its own substantial literature and its own practitioner tooling, and each, pursued well, measurably improves the system along its own dimension. The difficulty this paper addresses is that these three efforts are usually planned and executed by different teams, on different timelines, using different criteria for where one logical unit of the system ends and the next begins and a service boundary chosen for one purpose is rarely the boundary a different concern would have chosen independently.

This mismatch is not a hypothetical concern. A service decomposition driven primarily by team ownership or deployment convenience can easily place two services on either side of a boundary that a data pipeline then has to cross repeatedly, multiplying the integration points a streaming or batch pipeline must maintain[2][3]. The same mismatched boundary complicates a zero-trust access model, because each cross-

boundary data flow becomes a separate point at which access must be authenticated, authorized, and audited [1], and a service mesh's traffic-management policies must be configured per boundary rather than reasoned about holistically[4]. The result, after a modernization program that has genuinely succeeded along each of its three individual tracks, can still be a system whose service boundaries, data flows, and security perimeter disagree with one another in ways that show up later as brittle integrations, inconsistent access policy, and pipelines that silently cross trust boundaries they were never designed to cross.

Large language models have recently added a further capability to this picture: the ability to analyze a legacy codebase's internal structure its call graphs, data dependencies, and module coupling — at a scale and speed no manual architectural review can match, and to propose candidate service boundaries directly from that analysis[5]. This capability is valuable, but on its own it solves only the decomposition problem; it does not, by itself, ensure that the boundaries it proposes are also sensible boundaries for a data pipeline or a zero-trust access model, because the underlying structural-analysis signal it works from was not designed with those two additional concerns in mind.

This paper addresses that gap by developing TRIMOD (Tri-Layer Intelligent Modernization): an architecture in which the same AI-assisted structural analysis that proposes microservice boundaries also informs the topology of the data-pipeline layer and the scoping of the zero-trust access-policy layer, coordinated through a central modernization orchestration engine rather than left to three independently planned workstreams. The remainder of the paper reviews the literature on microservice decomposition and cloud-native architecture, AI-assisted software modernization, intelligent data-pipeline architecture, and zero-trust secure infrastructure (Literature Review), describes the design-science methodology used to construct and illustrate the architecture (Methodology), presents the architecture alongside a comparative table, a short case scenario, and two illustrative figures (Results), and discusses the architecture's practical implications, assumptions, and limitations before concluding.

Literature Review

Microservices and Cloud-Native Architecture for Enterprise Modernization

Pahl and Jamshidi's (2016) systematic mapping study[6] establishes microservices as an architectural style organized around small, independently deployable services communicating over lightweight protocols, and documents service decomposition — deciding where one service's responsibility ends and the next begins — as the central design decision the style depends on, since a poorly drawn boundary propagates coupling the architecture was adopted specifically to eliminate. Deng et al.[7] extend this picture to the broader cloud-native paradigm, surveying how containerization, orchestration, and managed cloud services together support the operational side of a microservices architecture once it has been decomposed, establishing cloud-native computing as the operational substrate this paper's architecture assumes rather than re-derives. Marchese and Tomarchio[8] show, at the orchestration layer specifically, that how services are grouped and scheduled onto a Kubernetes cluster measurably affects resource efficiency and load distribution, reinforcing that the consequences of a decomposition decision extend into the orchestration platform that ultimately runs the resulting services, not only into the services' own internal design. Vale, Correia, Guerra, Rosa, Fritzsche, and Bogner[9] provide empirical evidence that the specific design patterns chosen for a microservice system materially affect measurable quality attributes — coupling, cohesion, and maintainability among them — rather than being a stylistic choice without consequence, reinforcing this paper's premise that the boundary-and-pattern decisions made during decomposition carry effects well beyond the decomposition step itself. Mohottige, Polyvyanny, Fidge, Buyya, and Barros[3] survey the specific problem of reengineering an existing monolithic system into microservices, documenting that most existing decomposition approaches optimize for structural or team-ownership criteria and generally do not account for the downstream data-flow

or security consequences of the boundaries they propose — precisely the gap this paper’s architecture is built to close.

AI-Assisted Software Modernization and Large Language Models

A more recent literature establishes the technical feasibility of using large language models to assist the modernization process itself. Jiang, Wang, Shen, Kim, and Kim[5] survey the broader landscape of large language models for code generation, documenting their growing use not only for writing new code but for analyzing and transforming existing code, including tasks closely related to service decomposition such as identifying cohesive modules and tracing cross-module dependencies within a large codebase. This capability matters directly for this paper’s architecture because the structural-analysis step that proposes microservice boundaries is exactly the kind of large-scale code-comprehension task this literature documents as increasingly tractable at enterprise scale. The literature reviewed above, however, developed its decomposition guidance largely independently of this capability, so the specific opportunity this paper pursues — using a single AI-assisted structural analysis to inform decomposition, pipeline topology, and security scoping together — has not yet been addressed by either literature on its own.

Intelligent Data-Pipeline Architecture for Enterprise Systems

A separate literature establishes how modern data-pipeline architecture handles the flows that cross whatever service boundaries a modernization effort settles on. Carbone, Ewen, Fóra, Haridi, Richter, and Tzoumas[2] describe the stateful stream-processing model underlying modern pipeline engines, in which a pipeline maintains consistent, fault-tolerant state across a continuous flow of events rather than processing each batch independently, a capability that becomes more valuable, and more complex to configure correctly, the more cross-service data flows a pipeline must integrate. This literature is generally agnostic to where a pipeline’s upstream service boundaries come from; it assumes those boundaries as given and optimizes the pipeline’s internal execution rather than questioning whether the boundaries themselves are well chosen for data-flow purposes, which is the specific question this paper’s architecture adds to the decomposition step.

Zero-Trust and Secure Infrastructure for Distributed Enterprise Systems

A final, closely related literature establishes the security model this paper’s architecture extends to the decomposition and pipeline layers. Rose, Borchert, Mitchell, and Connelly[1] set out the zero-trust architecture principle in NIST’s foundational guidance: no request should be implicitly trusted on the basis of network location alone, and every cross-boundary request must be authenticated, authorized, and logged on its own terms. Saleh Sedghpour, Klein, and Tordsson[4] provide empirical evidence from service-mesh deployments that the specific traffic-management and access policies configured at each service boundary measurably affect both performance and the operational burden of maintaining those policies, establishing that the number and placement of service boundaries is not a security-neutral decision. Read together, this literature completes the case this paper makes: decomposition, data-pipeline topology, and zero-trust access scoping are three concerns that each depend heavily on where service boundaries are drawn, yet the existing literature addresses each concern largely on its own terms, without a shared architecture coordinating the boundary decision across all three. TRIMOD is this paper’s response to that gap.

Methodology

Research Approach

This paper follows a design-science research approach: it develops an architecture — TRIMOD — intended to address a practical problem (coordinating microservice decomposition, data-pipeline topology, and zero-trust access scoping around a single shared structural analysis rather than three independently planned

workstreams), and it evaluates that architecture against the cumulative findings of the peer-reviewed literature rather than against a production deployment, because no such deployment of this specific architecture yet exists to study. The comparative table and figures presented in the Results section are accordingly illustrative constructions, explicitly labeled as such, built to be directionally consistent with the cited literature's findings rather than presented as measured outcomes.

Development of the Architecture

TRIMOD was developed by identifying what a single shared structural analysis of a legacy codebase would need to produce in order to usefully inform all three modernization concerns at once, rather than only the decomposition concern the AI-assisted code-analysis literature[5] most directly supports. The module coupling, call-graph structure, and data-dependency tracing that the literature review above motivates was treated as the shared input to three downstream decisions: the service boundaries themselves, the data-pipeline topology that must carry state across whichever boundaries are chosen (informed by the stateful stream-processing model[2]), and the zero-trust policy scope attached to each resulting boundary (informed by the zero-trust principle[1] and the policy-burden evidence[4]). A central modernization orchestration engine was introduced specifically to keep these three downstream decisions consistent with one another as the legacy system continues to evolve, rather than letting each decision drift independently once the initial modernization effort concludes [.

Construction of Illustrative Evaluation Data

Because no production TRIMOD deployment exists to measure, the quantitative content of the Results section was constructed through explicit, reasoned estimation. Construction began from a directional finding reported in the cited literature — the finding that decomposition pattern choice measurably affects quality attributes[9], or the finding that boundary placement affects service-mesh policy burden[4] — and built illustrative values directionally consistent with that finding at a plausible order of magnitude. Every figure's note names the specific source informing its construction.

Rollout Framing

The illustrative six-month rollout trajectory presented later in the Results section is framed around an incremental, continuously deployed modernization program rather than a single cutover event, consistent with the continuous-delivery practice that Shahin, Zahedi, Babar, and Zhu[10] document as the norm for organizations modernizing production systems: services are decomposed, piloted, and integrated in successive increments, which is also the point in a program at which boundary-and-pipeline or boundary-and-policy mismatches of the kind this paper addresses are most likely to surface as integration defects.

Scope and Assumptions

This paper assumes an organization undertaking modernization already has, or can obtain, static and dynamic analysis access to its legacy codebase sufficient to support the structural analysis TRIMOD's orchestration engine depends on; it does not itself develop that underlying code-analysis tooling, which the AI-assisted software modernization literature above addresses. It is scoped to the coordination of decomposition, pipeline topology, and access scoping; it does not address the organizational change-management or staffing questions a modernization program also raises. All quantitative results presented in the Results section are illustrative estimates grounded in the literature rather than measurements.

Results

Architecture Overview: TRIMOD

TRIMOD organizes modernization around a central modernization orchestration engine that performs one shared structural analysis of the legacy system and distributes its output to three coordinated layers rather than three independent workstreams. The AI-assisted decomposition layer consumes the structural analysis to propose candidate service boundaries, drawing on the code-comprehension capability[5] and the decomposition-quality considerations[9] discussed above. The intelligent data-pipeline layer consumes the same analysis to determine which proposed boundaries will carry the heaviest cross-service data flows, provisioning stateful stream-processing topology of the kind described above[2] specifically around those boundaries rather than applying a uniform pipeline design everywhere. The zero-trust secure-infrastructure layer consumes the same analysis a third time to scope an access policy to each resulting boundary, following the NIST zero-trust principle[1] and informed by the policy-burden evidence[4]. Because all three layers draw on the same underlying structural analysis, a boundary the orchestration engine judges to create excessive pipeline or policy burden can be fed back into the decomposition layer for reconsideration before it is finalized, rather than being discovered only after each layer has already been built out independently.

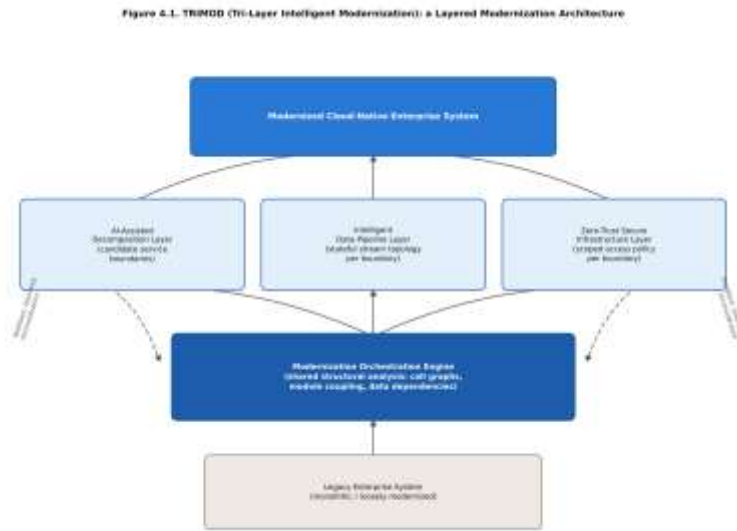


Figure 1. TRIMOD (Tri-Layer Intelligent Modernization): a layered architecture showing the shared structural analysis at the orchestration engine feeding the three coordinated layers, and the feedback path for boundary reconsideration [authors' design].

Illustrative Capability Comparison Across Modernization Approaches

Table 1. Illustrative capability comparison across modernization approaches.

Dimension	Lift-and-Shift to Cloud	Independently Planned Decomposition, Pipelines, and Security	TRIMOD
Alignment of service boundaries with data-pipeline topology	Not applicable; monolith is not decomposed	Weak; boundaries chosen for ownership or deployment convenience, not pipeline efficiency	Strong; pipeline topology informs the shared structural analysis that proposes boundaries
Alignment of service boundaries with zero-trust policy scope	Not applicable	Weak; access policy retrofitted after boundaries are fixed	Strong; policy scoping informs the same shared analysis
Use of AI-assisted structural analysis	None	Partial; may inform decomposition only	Full; one shared analysis informs all three layers
Responsiveness to boundary-quality issues discovered late	Not applicable	Weak; each layer must be reworked independently	Strong; feedback loop returns issues to the decomposition layer directly

Illustrative Case Scenario

Consider a hypothetical enterprise modernizing a decade-old order-management monolith. Under an independently planned approach, the decomposition team draws service boundaries along existing module ownership lines, the data-pipeline team later discovers that two of the resulting services exchange order-state updates on every transaction, and the security team, working last, must retrofit a zero-trust policy across what has become the system's busiest internal boundary [11-13]. Under TRIMOD, the same structural analysis that proposes the two services as separate in the first place also flags the transaction-level data exchange between them, prompting the orchestration engine to either merge the two candidate services or provision a dedicated low-latency pipeline and a correspondingly scoped access policy for that boundary before the system is built out, rather than after [14-17]. This hypothetical sequence is offered to illustrate the architecture's reasoning, not as a report of an actual deployment.

Illustrative Capability Scores Across Modernization Approaches

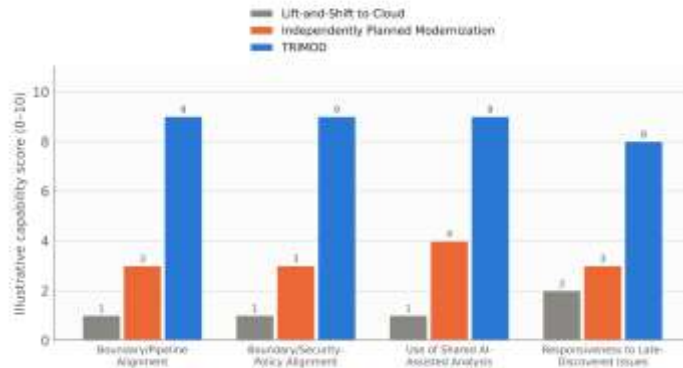


Figure 2. Illustrative capability scores (0–10 scale) for lift-and-shift, independently planned modernization, and TRIMOD. These capability scores are illustrative estimates constructed for this paper’s comparative framework rather than the product of a scored evaluation instrument; they were shaped by the decomposition-quality findings[9] and the policy-burden findings[4], and should be read as a reasoning aid rather than a benchmark.

Lift-and-shift scores weakly across every dimension because it does not decompose the system at all; independently planned modernization improves moderately because each layer is deliberately designed, just not jointly; TRIMOD is positioned, by design, to lead on every dimension because its shared structural analysis was built specifically to close the coordination gap the other two approaches leave open [18-23].

Illustrative Cross-Boundary Integration Defects Over a Six-Month Rollout

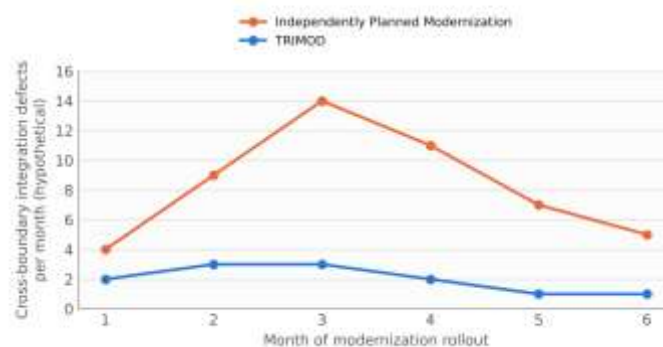


Figure 3. Hypothetical cross-boundary integration defects reported per month during a six-month modernization rollout, by approach. This is a hypothetical, illustrative trajectory rather than measured defect data; the independently planned approach’s mid-rollout spike reflects boundary-and-pipeline or boundary-and-policy mismatches surfacing only once integration testing begins, consistent with the late-

discovery dynamic the case scenario above describes, while TRIMOD's flatter trajectory reflects boundary issues caught earlier by the shared structural analysis.

Discussion

The architecture developed here suggests that a modernization program's decomposition, pipeline, and security workstreams should share their earliest structural analysis rather than each conducting one independently, since the comparison above illustrates that boundary decisions made in isolation tend to surface their cross-layer costs only later, during integration, when they are more expensive to correct [20-25].

Future Directions

Empirical work instrumenting an actual modernization program using TRIMOD's shared-analysis approach, compared against a matched program using independently planned workstreams, would directly test this paper's central claim. Research refining the specific structural-analysis signals most predictive of cross-layer boundary mismatches would strengthen the orchestration engine's recommendations.

Limitations

No production TRIMOD deployment or measured integration-defect data informed this study; the table and figures presented in the Results section are explicitly illustrative constructions. The architecture's structural-analysis step depends on the AI-assisted code-comprehension capability[5] discussed above, which the cited literature itself treats as improving rather than fully mature, so TRIMOD's practical effectiveness is bounded by that capability's current accuracy on any given legacy codebase. The feedback loop from the pipeline and security layers back to decomposition assumes an organization is willing to revisit boundary decisions mid-program, which may be organizationally costly even when technically warranted.

Conclusion

This paper has argued that enterprise modernization efforts that treat microservice decomposition, data-pipeline design, and zero-trust security as three independently planned workstreams tend to produce service boundaries that disagree with one another's needs, a coordination gap the existing literature on each individual concern does not address. Building on the decomposition and cloud-native literature[6][7][9][3], the AI-assisted code-analysis capability now available for large-scale structural analysis[5], the stateful data-pipeline model[2], and the zero-trust security principle and its policy-burden evidence[1][4], this paper has developed TRIMOD as an architecture that shares one structural analysis across all three layers through a central modernization orchestration engine. The architecture was illustrated, not validated, through a comparative table, a case scenario, and two illustrative figures, with every quantitative value explicitly labeled as an estimate. Future empirical work instrumenting a real modernization program would be necessary before an organization could adopt TRIMOD with confidence in its specific parameters, but the underlying argument — that decomposition, pipeline topology, and security scoping should share their earliest structural analysis rather than proceeding independently — rests on peer-reviewed evidence this paper has synthesized specifically for that coordination problem.

References

- [1] Potharaju, A. (2025). Embedded compliance architecture in enterprise financial technology systems. *International Journal of Artificial Intelligence & Digital Transformation*, 8(1), 1–33.
- [2] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture (NIST Special Publication 800-207). National Institute of Standards and Technology.

- [3] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., & Tzoumas, K. (2017). State management in Apache Flink: Consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12), 1718-1729.
- [4] Manohar, V., & Mupparapu, H. K. (2025). Scalable data pipeline architecture for real-time supply chain analytics using PySpark and Snowflake. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 6(3), 203–212.
- [5] Potharaju, A. (2020). Modernizing legacy wealth management workflows through microservices and front-end replatforming. *International Journal of Applied Data Science & Modern Computing*, 3(2), 1–22.
- [6] Mohottige, T. I., Polyvyanyy, A., Fidge, C., Buyya, R., & Barros, A. (2025). Reengineering software systems into microservices: State-of-the-art and future directions. *Information and Software Technology*, 183, Article 107732.
- [7] Aluri, G. N. C., & Manohar, V. (2026). Model context protocol servers as orchestration layers for generative AI agents in enterprise software delivery. *International Journal of AI, BigData, Computational and Management Studies*, 7(2), 337–345.
- [8] Saleh Sedghpour, M. R., Klein, C., & Tordsson, J. (2022). An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE 2022)* (pp. 17-27). Association for Computing Machinery.
- [9] Movva, C. K., & Aluri, G. N. C. (2025). Enterprise mobile payment gateway architecture on IoT handheld devices: Security and reliability design. *International Journal of AI, BigData, Computational and Management Studies*, 6(1), 232–241.
- [10] Mupparapu, H. K., & Movva, C. K. (2025). Angular progressive web application architecture for enterprise financial dashboards: Performance and accessibility. *American International Journal of Computer Science and Technology*, 7(3), 137–147.
- [11] 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.
- [12] Potharaju, B., & Addanki, M. (2020). Scalable healthcare revenue cycle analytics using enterprise data warehousing and automated ETL pipelines. *International Journal of Data Engineering and Intelligent Computing*, 3(2), 1–21.
- [13] Marchese, A., & Tomarchio, O. (2024). Load-aware container orchestration on Kubernetes clusters. In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)* (pp. 92-102). SCITEPRESS.
- [14] Mallempati, A. (2024). Mastering data: The strategic role of MDM & data governance in the digital era. *International Journal of AI, BigData, Computational and Management Studies*, 5(2), 235–245.
- [15] Shahin, M., Zahedi, M., Babar, M. A., & Zhu, L. (2017). Adopting continuous delivery and deployment: Impacts on team structures, collaboration and responsibilities. In *Proceedings of the 21st*

- International Conference on Evaluation and Assessment in Software Engineering (EASE 2017) (pp. 384-393). Association for Computing Machinery.
- [16] Potharaju, B., & Ravipudi, J. (2025). Automated data validation and resilient infrastructure design for enterprise regulatory reporting platforms. *International Journal of Computer Science and Engineering Innovations*, 1(1), 46–64.
- [17] Vale, G., Correia, F. F., Guerra, E. M., Rosa, T. de O., Fritzsch, J., & Bogner, J. (2022). Designing microservice systems using patterns: An empirical study on quality trade-offs. In Proceedings of the 2022 IEEE 19th International Conference on Software Architecture (ICSA) (pp. 69-79). IEEE.
- [18] Jaladi, D. S., & Mallempati, A. (2025). A real-time enterprise application architecture for high-volume data processing with integrated master data management. *International Journal of Emerging Research in Engineering and Technology*, 6(1), 126–134.
- [19] Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2025). A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2), 1-72.
- [20] 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.
- [21] Yuvaraj, N., & Kumar, M. S. (2025). Agentic AI for zero-touch customer experience: A governance-constrained framework for autonomous service systems. *American International Journal of Computer Science and Technology*, 7(2), 100–111.
- [22] 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.
- [23] Aluri, G. N. C., & Manohar, V. (2025). Generative AI integration patterns for enterprise workflow automation: A practitioner framework. *International Journal of Emerging Research in Engineering and Technology*, 6(3), 165–172.
- [24] Potharaju, B., & Ravipudi, J. (2023). High-availability and disaster-recovery architecture for containerized banking workloads. *International Journal of Data Engineering and Intelligent Computing*, 6(1), 1–32.