Vishnu Vardhan Reddy Kavuluri[1], Jaswanth Kumar Mandapatti[2], Srinivasarao Bandla[3], Nareshkumar Jagadhabi[4], Maheswara Rao Gorumutchu[5]
[1]Deloitte Consulting LLP, United States
[2]Advent Health, United States
[3]Deloitte Consulting LLP, United States
[4]Compnova Inc, United States
[5]HYR Global Source Inc, United States

Received: 08-08-2025; Revised: 16-09-2025; Accepted: 28-10-2025

Entanglement Effects in Auto-Generated Enterprise Application Artifacts

Abstract
Version entanglement in auto-generated enterprise application artifacts poses a significant challenge as dependencies across multiple versions become increasingly complex and tightly coupled. Existing approaches focus primarily on version control and artifact generation efficiency, but they often overlook the systemic impact of cross-version interactions and propagation effects. This study investigates how dependencies between generated artifacts create entangled structures that amplify change propagation across layers such as code, configuration, and services. The analysis demonstrates that dependency density, centrality of key versions, and cross-layer interactions drive non-linear growth in entanglement, leading to instability and reduced maintainability. By modeling version interactions as a network and identifying critical propagation patterns, the study provides insights into controlling entanglement through modular design, dependency isolation, and structured generation strategies. The findings contribute to improving reliability and scalability in enterprise application lifecycle management.

Keywords: Version Entanglement, Auto-Generated Artifacts, Dependency Networks, Enterprise Systems.

1. Introduction

Version entanglement has become a critical issue in modern enterprise application ecosystems where artifacts are increasingly auto-generated through model-driven and low-code development platforms. These artifacts include code modules, configuration files, service definitions, and integration layers that evolve across multiple versions simultaneously. As systems grow, dependencies between versions become tightly coupled, leading to complex interaction patterns that are difficult to manage [1]. This entanglement introduces challenges in maintaining consistency and traceability across releases [2]. Changes in one version often propagate unexpectedly to others. This creates instability in deployment and execution. Understanding version interactions is essential for reliable system evolution.

Auto-generated enterprise artifacts are typically produced through automated pipelines that translate high-level models into executable components. While this improves development speed, it also introduces hidden dependencies between generated artifacts. These dependencies are often not explicitly documented, making them difficult to track and manage [3]. As multiple versions of artifacts coexist, their interactions create overlapping dependency chains. This leads to situations where changes in one artifact version affect others indirectly [4]. The complexity increases with scale and frequency of updates. This creates a dense network of interdependent components. Managing such systems becomes increasingly challenging.

The concept of version entanglement is closely related to dependency coupling in software systems, where tightly coupled components share execution and data relationships. In auto-generated environments, this coupling is amplified due to shared templates, code generators, and configuration models [5]. Even minor modifications in generation logic can affect multiple artifact versions simultaneously. This results in cascading changes across the system. The lack of clear boundaries between versions further complicates the issue. As a result, isolating the impact of changes becomes difficult. This contributes to reduced system maintainability.

Another contributing factor is the reuse of common templates and schemas across different versions of artifacts. While reuse improves efficiency, it creates shared dependencies that link multiple versions together. Changes to a shared template can propagate across all derived artifacts, leading to widespread impact [6]. This creates a situation where version independence is compromised. The system becomes sensitive to changes in shared components [7]. This increases the risk of unintended side effects. Such propagation patterns are a key characteristic of version entanglement.

Enterprise systems often involve multiple layers, including application logic, middleware, and integration services, each with its own versioning strategy. When these layers interact, version mismatches and compatibility issues can arise. Studies on enterprise system integration highlight that cross-layer dependencies significantly increase system complexity [8]. These interactions create multi-dimensional entanglement across layers. A change in one layer can trigger adjustments in others. This

leads to synchronization challenges during deployment. Managing compatibility becomes a critical concern.

Continuous integration and deployment pipelines further amplify version entanglement by introducing frequent updates and rapid iteration cycles. Automated builds and deployments generate new artifact versions at a high rate, increasing the number of interacting versions in the system. Research on DevOps and continuous delivery systems shows that high update frequency can lead to dependency conflicts and integration issues [9, 10]. These rapid changes make it difficult to maintain a stable system state. Version tracking becomes more complex over time. This increases the likelihood of entanglement-related failures.

Despite the growing importance of this issue, there is limited research that systematically analyzes version entanglement in auto-generated enterprise artifacts. Most existing work focuses on version control and dependency management at the code level, without addressing the unique challenges of generated artifacts. Recent studies have begun to explore the impact of automated generation on system complexity, but a comprehensive framework is still lacking [11]. This gap highlights the need for deeper analysis. Understanding entanglement effects is essential for improving system reliability. It also supports better design of generation pipelines.

This research focuses on analyzing how version entanglement emerges in auto-generated enterprise artifacts and how it affects system stability and maintainability. It examines the role of shared dependencies, generation logic, and cross-layer interactions in creating entangled structures. The study aims to identify patterns of entanglement and propose ways to manage them effectively. By understanding these dynamics, systems can be designed to reduce unintended interactions. This supports more predictable and stable deployments. The findings contribute to improving enterprise application lifecycle management.

2. Methodology

The methodology is designed to analyze version entanglement effects in auto-generated enterprise application artifacts by modeling how dependencies evolve and interact across multiple versions. The system is treated as a multi-version dependency network, where each artifact version is represented as a node and interactions between versions are represented as edges. This approach allows systematic tracking of how changes propagate across versions. The focus is on identifying patterns of coupling and interaction that lead to entanglement. The methodology integrates structural modeling with dependency analysis. It aims to quantify the strength and spread of version interactions. This provides a clear framework for studying entanglement dynamics.

Let the set of artifact versions be denoted as $V = \{v_1, v_2, \ldots, v_n\}$, where each $v_i$represents a generated artifact at a specific version. Dependencies between versions are represented using an adjacency matrix D, defined as

$$D_{ij} = \begin{cases} 1, & \text{if version } v_i \text{ depends on } v_j \\ 0, & \text{otherwise} \end{cases}$$

This matrix captures direct dependency relationships between versions. It forms the basis for analyzing interaction patterns. The density of this matrix reflects the level of entanglement. Higher density indicates stronger coupling between versions.

To measure entanglement intensity, the methodology introduces an interaction coefficient defined as

$$E_i = \sum_{j=1}^{n} D_{ij}$$

which represents the number of dependencies associated with version v_i. A higher value of $E_i$indicates that a version is highly interconnected with others. This increases the likelihood of cascading effects during updates. The distribution of $E_i$across all versions provides insight into system-wide entanglement. It helps identify critical versions that act as hubs. These hubs play a key role in propagation behavior.

The methodology also defines a propagation function to model how changes spread across versions. If a change originates in version v_k, the propagation effect is expressed as

$$P(v_k) = \sum_{i=1}^{n} (D^m)_{ki}$$

where $m$represents the number of propagation steps. This formulation captures indirect dependencies through multi-step interactions. It allows analysis of how deeply a change can affect the system. Larger values of $P(v_k)$indicate wider impact. This helps identify versions with high propagation influence.

A layered artifact model is constructed to represent different types of generated components, such as code modules, configuration files, and service interfaces. Each layer may have its own versioning and dependency structure. Interactions between layers introduce cross-layer dependencies, which contribute significantly to entanglement. The methodology captures these interactions by extending the dependency matrix to include layer-specific relationships. This enables analysis of both intra-layer and inter-layer dependencies. It provides a more complete view of system complexity. Layered modeling helps isolate sources of entanglement.

To simulate realistic conditions, the methodology includes version evolution scenarios where artifacts are updated incrementally over time. Each update introduces new dependencies or modifies existing ones. The system is observed under multiple update cycles to analyze how entanglement evolves. This dynamic analysis reveals patterns of growth in dependency density. It also highlights points where the

system becomes highly interconnected. Such scenarios reflect real-world enterprise environments. They help evaluate the long-term impact of version interactions.

The methodology incorporates metrics for evaluating system stability, including dependency density, propagation depth, and version centrality. Dependency density measures the proportion of existing dependencies relative to all possible connections. Propagation depth indicates how far changes can spread across versions. Version centrality identifies nodes that play a dominant role in dependency networks. These metrics provide a quantitative basis for analyzing entanglement. They help identify critical areas within the system. Monitoring these metrics supports effective system management.

To account for mitigation strategies, the methodology evaluates techniques such as dependency isolation, version encapsulation, and modular generation. Dependency isolation reduces direct interactions between versions. Version encapsulation ensures that changes are contained within specific boundaries. Modular generation separates artifact creation into independent units. These strategies are incorporated into the model to observe their effect on entanglement. Comparative analysis is performed to measure improvement. This helps determine effective approaches for reducing complexity.

All dependency relationships and interaction strengths are summarized in Table 1, which presents a cross-version dependency interaction matrix. This matrix provides a compact representation of how versions influence each other. Each cell indicates the presence and intensity of interaction between two versions. The matrix format allows easy identification of highly connected regions. It also highlights asymmetric dependencies. This structured representation supports detailed analysis of entanglement patterns. It serves as a reference for interpreting system behavior.

Table 1. Cross-Version Dependency Interaction Matrix for Auto-Generated Enterprise Artifacts

Version	v1	v2	v3	v4	v5
v1	0	1	0	1	0
v2	1	0	1	0	1
v3	0	1	0	1	1
v4	1	0	1	0	1
v5	0	1	1	1	0

3. Results and Discussion

The results reveal that version entanglement emerges progressively as dependencies accumulate across auto-generated artifacts. In the initial stages, the dependency network remains relatively sparse, with limited interactions between versions. Changes introduced in one version affect only a small subset of related artifacts. However, as the number of generated versions increases, the dependency structure becomes denser. This leads to a rise in cross-version interactions and shared dependencies. The system begins to exhibit interconnected clusters of artifacts. These clusters act as localized regions of entanglement. This marks the transition from isolated dependencies to systemic coupling.

As dependency density increases, the propagation of changes becomes more widespread and less predictable. A modification in a single version can trigger a cascade of updates across multiple connected versions. The results show that propagation depth grows rapidly with increasing connectivity. This indicates that indirect dependencies play a significant role in entanglement behavior. Even versions with no direct connection can be affected through multi-step interactions. This amplifies the impact of changes across the system. The network becomes highly sensitive to small modifications. Such behavior increases the risk of unintended system-wide effects.

Figure 1 illustrates the version entanglement propagation network across artifact layers using a directed graph representation. Nodes represent artifact versions across layers such as code, configuration, and services, while edges indicate dependency relationships. The graph shows the formation of dense clusters where multiple versions are interconnected. These clusters highlight regions of high entanglement. The direction of edges indicates the flow of dependency and propagation paths. Some nodes emerge as central hubs with a high number of incoming and outgoing connections. These hubs play a critical role in spreading changes across the network.

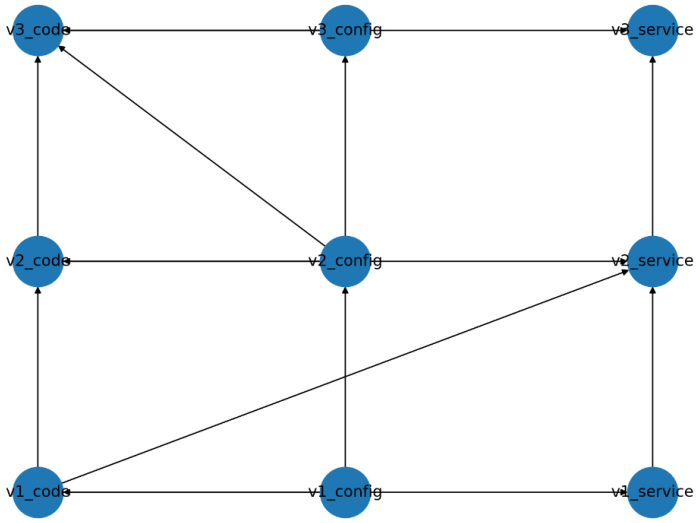

Figure 1. Version Entanglement Propagation Network Across Artifact Layers

The analysis of network structure reveals that certain versions act as critical propagation points due to their high centrality. These versions are typically derived from shared templates or frequently updated components. Changes in such nodes result in widespread impact across multiple layers. The results show that removing or isolating these nodes significantly reduces overall entanglement. This indicates that targeted interventions can effectively control propagation behavior. Identifying such critical nodes is essential for managing system complexity. It provides a practical approach to mitigating entanglement risks.

Cross-layer interactions further intensify entanglement by linking artifacts from different functional domains. For example, changes in configuration artifacts can propagate to code modules and service interfaces. This creates multi-layer dependency chains that increase propagation complexity. The results show that cross-layer edges contribute more to propagation depth than intra-layer connections. This highlights the importance of managing interactions between layers. Without proper isolation, these interactions can lead to widespread instability. Cross-layer entanglement is therefore a major factor in system behavior.

Temporal analysis of version updates shows that entanglement grows cumulatively over time. Each new version introduces additional dependencies, which build upon existing structures. The network becomes increasingly dense with each update cycle. This leads to a gradual increase in propagation risk and system complexity. The results indicate that early-stage design decisions have long-term effects on entanglement. Systems that start with high dependency density tend to become more complex over time. Managing dependencies early is therefore critical.

4. Conclusion

The study demonstrates that version entanglement is an inherent and growing challenge in auto-generated enterprise application artifacts, driven by dense dependency structures and repeated reuse of generation logic. As systems evolve, interactions between artifact versions become increasingly complex, leading to tightly coupled networks that are difficult to manage. The results confirm that even small changes in one version can propagate widely across multiple artifacts, creating instability and reducing predictability. This highlights that entanglement is not an isolated issue but a systemic characteristic of modern enterprise platforms.

A key finding of this work is the identification of structural patterns that contribute to entanglement, particularly high centrality nodes and cross-layer dependencies. Versions derived from shared templates or frequently updated components act as propagation hubs, amplifying the spread of changes. Cross-layer interactions further intensify this effect by linking artifacts across functional domains such as code, configuration, and services. These insights suggest that controlling entanglement requires targeted interventions rather than broad system-wide changes. Isolating critical nodes and managing cross-layer dependencies can significantly reduce propagation risks.

The analysis also emphasizes the importance of proactive design strategies in mitigating entanglement. Techniques such as modular artifact generation, version encapsulation, and dependency isolation can help limit the growth of interaction networks. By reducing unnecessary coupling, systems can maintain clearer boundaries between versions and improve maintainability. Additionally, monitoring dependency density and propagation depth provides early indicators of entanglement growth. This enables timely corrective actions before the system becomes overly complex.

In conclusion, addressing version entanglement requires a combination of structural awareness and systematic control mechanisms within enterprise application platforms. The findings provide a foundation for designing more scalable and maintainable systems by minimizing unintended interactions between artifact versions. Future work can explore automated detection of entanglement patterns and adaptive strategies for managing dependencies in real time. By improving control over version interactions, organizations can achieve more stable deployments and efficient lifecycle management in auto-generated environments.

References

1. Siame, A., & Kunda, D. (2017). Evolution of PHP applications: A systematic literature review. *International Journal of Recent Contributions from Engineering, Science & IT (iJES)*, *5*(1), 28-39.
2. Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
3. Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice*. Morgan & Claypool Publishers.
4. Verbruggen, C., & Snoeck, M. (2023). Practitioners' experiences with model-driven engineering: a meta-review. *Software and Systems Modeling*, *22*(1), 111-129.
5. Bontchev, B., & Milanova, E. (2020). On the usability of object-oriented design patterns for a better software quality. *Cybernetics and Information Technologies*, *20*(4), 36-54.
6. Cervantes, H., & Kazman, R. (2024). *Designing software architectures: a practical approach*. Addison-Wesley Professional.
7. Cervantes, H., & Kazman, R. (2024). *Designing software architectures: a practical approach*. Addison-Wesley Professional.
8. Hameed Ahmad, G. T. (2021). Seamless Enterprise Integration: Cloud-Native Strategies for the Modern Era.
9. James, M. (2025). Enhancing Software Reliability: The Role of Automated Continuous Integration and Continuous Delivery.
10. Mousaei, M. A. H. D. I., & Gandomani, T. J. (2020). DevOps Approach and Lean Thinking in Agile Software Development: Opportunities, Advantages, and Challenges. *J. Soft Eng. Int. Syst*, *5*, 1-10.
11. Akthar, S. R., Islam, M. R., Hasan, M. B., Siddiqua, M. S., Haque, S. I., Saad, J. A., ... & Hasan, M. (2024). Overcoming Obstacles in Model-Driven Engineering: Lessons from the Software Industry. In *ICSOFT* (pp. 153-160).