VECTORAL PUBLISHING

Research Article: Journal of Computational Social Dynamics Volume: 08

AI-Driven Partitioning Framework for Migrating Monolithic Applications to Microservices

Vijay Ramamoorthi

Independent Researcher

Abstract

Microservice Architecture (MSA) has become a standard for designing scalable and flexible enterprise applications. However, the process of migrating monolithic systems to microservices is fraught with challenges, including the risk of creating distributed monoliths and managing complex distributed transactions. To address these issues, we present **CARGO** (Context-sensitive lAbel pRopaGatiOn), an AI-guided framework that improves microservice partitioning through the use of **System Dependency Graphs (SDGs)** and a context-sensitive label propagation algorithm. SDGs enable detailed modeling of both intra-service and inter-service dependencies, capturing critical aspects like call-return, data, heap, and transactional edges. By iteratively refining partition boundaries, CARGO minimizes inter-service coupling, reduces the occurrence of distributed transactions, and enhances service cohesion. Evaluations conducted on benchmark applications such as Daytrader and JPetStore demonstrate that CARGO significantly outperforms state-of-the-art tools like Mono2Micro and CoGCN in terms of transaction purity, latency, throughput, and architectural quality. This work establishes a foundation for automating the migration of monolithic applications into efficient, scalable microservice architectures and opens avenues for applying CARGO to other programming environments like Python and .NET.

Keywords: *Microservices, Microservice architecture, Neuromorphic AI integration, CARGO framework System Dependency Graphs*

Introduction

Microservice architecture (MSA) has emerged as a transformative approach for the design and development of enterprise applications, offering significant benefits over traditional monolithic architectures. By decomposing a system into a set of loosely coupled, independently deployable services, MSA enhances scalability, facilitates agile development, and improves fault isolation. Each microservice can evolve independently, be deployed separately, and scale according to the specific needs of the functionality it provides. For large-scale, cloud-native applications, microservice architecture is particularly advantageous, allowing organizations to meet the growing demands of flexibility, performance, and rapid iteration in software development [1], [2].

Despite the advantages of MSA, migrating legacy monolithic applications to a microservice-based architecture presents substantial challenges. Monolithic systems typically consist of highly interdependent components, where functionalities and data flows are intertwined. Manually partitioning these systems into microservices is a complex, time-consuming process that is prone to error [3]. One of the significant risks during this migration process is the creation of a distributed monolith—an architecture in which services are deployed independently but remain tightly coupled through frequent inter-service communication, resulting in performance inefficiencies and operational complexity. Distributed monoliths often negate the intended benefits of microservices by introducing new bottlenecks in terms of scalability, fault tolerance, and maintainability. Another key challenge in microservice partitioning is the management of transactional boundaries [4]. Monolithic applications often rely on centralized, transactional databases, where consistency and integrity are maintained through strong ACID guarantees. When transitioning to microservices, maintaining these guarantees across distributed services can necessitate the use of distributed transactions, such as two-phase commits. Distributed transactions are not only complex to implement and manage, but they also introduce performance overheads and potential consistency issues. Thus, one of the critical objectives in effective microservice partitioning is to minimize the need for such transactions while preserving the integrity of business processes.

Traditional approaches to microservice partitioning rely on program analysis techniques such as call-graph and control-flow graph representations, which identify functional boundaries based on the interaction patterns within the code. These techniques, while useful, are often limited by their inability to account for critical dependencies, such as database interactions and indirect method calls. As a result, these methods may produce partitioning recommendations that either create distributed monoliths or require extensive use of distributed transactions [5], [6].

To address the limitations of traditional microservice partitioning methods, we propose the use of System Dependency Graphs (SDGs), which provide a more detailed and expressive representation of an application's structure by capturing not only call-return and control-flow relationships but also data, heap, and transactional dependencies. This allows for a more accurate modeling of complex inter-component relationships in monolithic applications, resulting in more effective microservice partitioning. We introduce CARGO (Context-sensitive lAbel pRopaGatiOn), an AI-driven framework that utilizes a novel context-sensitive label propagation algorithm to iteratively refine partitioning recommendations, minimizing inter-service coupling and addressing transactional dependencies. By preventing the creation of distributed monoliths and reducing the need for distributed transactions, CARGO significantly improves performance and maintainability. Our evaluation on benchmark Java applications like Daytrader and JPetStore shows that CARGO outperforms existing tools such as Mono2Micro and CoGCN, yielding higher cohesion, reduced coupling, improved throughput, lower latency, and greater transactional purity, positioning it as a robust solution for automating the migration of monolithic applications to scalable microservice architectures.

Background

Microservice partitioning has gained substantial attention due to its role in improving scalability, flexibility, and performance in distributed systems. This section delves into traditional and modern partitioning approaches, emphasizing the transition from call-graph-

based methods to system dependency graphs (SDGs). We also highlight advancements in partitioning strategies that address transactional and performance challenges in microservice architectures.

Traditional microservice partitioning methods, such as call-graph and control-flow-based techniques, were originally designed for monolithic applications. These methods rely heavily on control flow and data dependencies to segment applications into manageable components. Control-flow graphs, for example, have been widely used in embedded systems to accelerate specific portions of code through hardware optimizations [7]. While these approaches have been instrumental in partitioning applications, they face challenges when applied to microservice architectures due to the high coupling between components, leading to inefficient partitioning and increased latency in distributed systems [8]. To address this, newer methods, such as program slicing and dependency analysis, have emerged. These methods aim to create finer partitions by analyzing data flows and control dependencies at a deeper level [9]. However, these techniques often struggle with scaling to large applications and managing distributed transactions effectively.

SDGs have been introduced as an evolution of traditional dependency graphs, allowing for a more detailed representation of an application's structure. SDGs provide a more comprehensive view of both intra- and inter-service dependencies, enabling more efficient microservice partitioning [10]. In the context of microservice architectures, SDGs map the interactions between services, facilitating the identification of critical services and their dependencies [11]. This reduces the risk of creating distributed monoliths—systems that appear to be microservices but exhibit the same coupling issues as monoliths, resulting in suboptimal performance [12]. Research on SDG-based partitioning has shown promising results in optimizing both system performance and scalability. For instance, the CARGO framework demonstrated significant improvements in reducing transactional overheads and increasing throughput in Java EE applications [10]. This methodology also extends to optimizing system performance by automating transaction management in distributed systems [13].

Recent advancements in partitioning methods have incorporated AI-driven approaches to refine the process further. Machine learning algorithms and context-sensitive analysis are used to adapt partitions dynamically based on system demands and network conditions [14]. These techniques allow for more efficient resource utilization and minimize downtime during partitioning, particularly in large-scale distributed systems [15] Furthermore, studies have shown that applying dependency logging techniques in multi-node systems can significantly improve the efficiency of transaction management, reducing the complexity of distributed transactions and enhancing fault tolerance [16]. In this context, partitioning methods are increasingly relying on graph-based analysis to improve the accuracy and efficiency of the partitioning process [17] These advancements offer a glimpse into the future of microservice partitioning, where automated, AI-driven approaches will likely become the standard for optimizing the performance and scalability of distributed systems [18]. In addition to these, advancements in **neuromorphic AI integration** in distributed systems are also contributing to partitioning techniques. Neuromorphic systems, with their brain-inspired architecture, offer a highly resource-efficient solution to handle the increasing complexity in data generation from cyber-physical systems. The integration of AI-driven approaches to enhance resource allocation, dependency analysis, and performance efficiency in distributed systems. One of the

notable advancements is the **CARGO framework**, which introduces a context-sensitive label propagation algorithm designed to refine the partitioning quality of monolithic applications into microservices [10]. CARGO has proven to improve partition quality by significantly reducing distributed transactions and lowering system latency. Similarly, **dependency-aware microservice deployment** and **resource allocation** approaches have gained traction, optimizing deployment strategies based on interaction dependencies between microservices, which further reduces computation and transmission delays in distributed edge networks.

Methodology

This section presents the CARGO framework and its core algorithm, context-sensitive label propagation (LPA), for enhancing the partitioning of monolithic applications into microservices. CARGO leverages a System Dependency Graph (SDG) to model the intricate relationships between different components of the system, including data, heap, control, and transactional dependencies. The methodology is structured in three key phases: (i) the construction of the SDG, (ii) context-sensitive label propagation (LPA), and (iii) refinement and evaluation of microservice partitioning.

System Dependency Graph Construction

The SDG is a directed graph $G = (V, E)$, where *V* represents the set of nodes (such as classes, methods, and database entities), and *E* represents the set of edges (such as call-return, data dependencies, heap dependencies, and transactional interactions). The SDG captures both intra- and inter-procedural relationships, enabling a more detailed representation of the system than traditional call graphs or control-flow graphs.

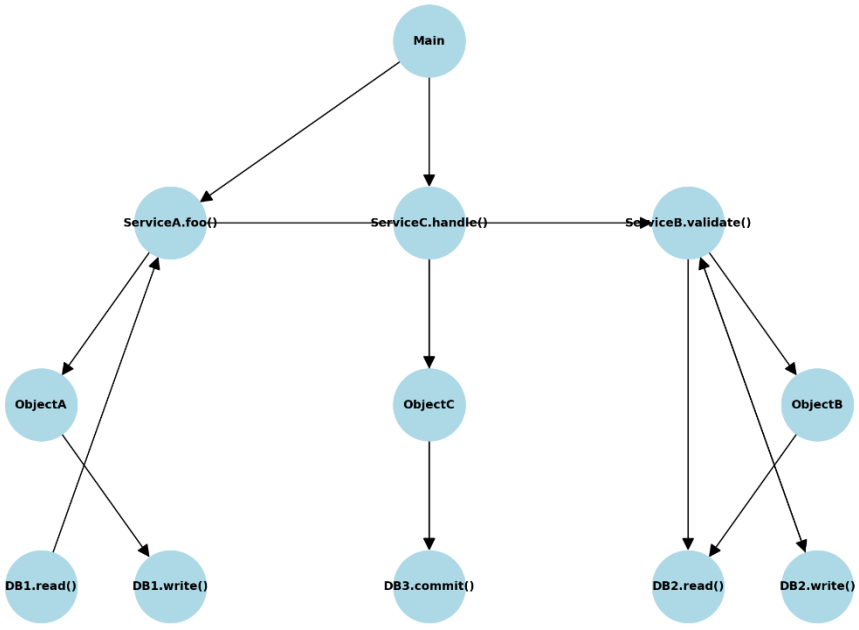

Figure 1 System Dependency Graph (SDG) illustrating Application, Middleware, and Data layers. The graph depicts the interactions between services (ServiceA, ServiceB, and ServiceC) and their corresponding data access points (DB1, DB2, DB3). Each service interacts with objects or database operations, showcasing the flow of dependencies within the system. This visualization is useful for identifying potential bottlenecks and understanding the system's transactional flow.

Nodes and Edges

- Nodes *V* Nodes represent various entities within the application, including:

- Methods (*M):* Individual functions or procedures within the system.

- Database Tables (*D)*: Representations of transactional dependencies between services.

- Heap Objects (*H*): Objects allocated and shared between different components.

- Edges *E*: Different types of edges represent the various relationships between the nodes:

- Call-Return Edges E_{call} : Denote method invocations between different classes or services.

- Data Dependencies E_{data} : Capture the flow of data between different methods and database interactions.

- Heap Dependencies E_{heap} : Represent shared objects or references passed between methods or classes.

- Transactional Dependencies E_{trans} : Capture interactions between methods and database tables, representing transactional operations like reads or writes. Mathematically, the SDG can be represented as:

$$G = (V, E) \quad \text{where} \quad V = M \cup D \cup H, \quad E = E_{\text{call}} \cup E_{\text{data}} \cup E_{\text{heap}} \cup E_{\text{trans}} \tag{1}$$

Each edge is qualified with a context-sensitive identifier, distinguishing different invocations of the same method depending on the context in which it is called.

Context-Sensitive Label Propagation (LPA)

Once the SDG is constructed, the next step is to apply the context-sensitive label propagation algorithm to identify optimal partitions. The goal of LPA is to group nodes that are functionally related and minimize inter-service coupling while maximizing intra-service cohesion.

Label Propagation Algorithm

The basic principle behind label propagation is that nodes within the SDG that are strongly connected (i.e., share a significant number of edges) should belong to the same microservice partition. Label propagation is performed iteratively, where each node updates its label based on the labels of its neighbors until convergence. Define $l(v)$ as the label of node *v*, representing its partition assignment. Each node $v \in V$ iteratively updates its label to match the most frequent label among its neighbors $N(v)$. Mathematically, this update rule can be expressed as:

$$l(v) = \arg\max_{l'} \sum_{u \in N(v)} \delta(l(u), l') \tag{2}$$

where $\delta(x, y)$ is the Kronecker delta function, returning 1 if $x = y$ and 0 otherwise. This process is repeated until the labels converge, meaning no node changes its label during an iteration.

Context Sensitivity

In CARGO, the label propagation is context-sensitive, meaning the relationships between nodes are qualified by their specific contexts. Let $C(v,u)$ represent the context in which an edge $(v,u) \in E$ exists. For example, the context may represent different states of the program at runtime or different invocation paths of a method. The label propagation algorithm is extended to operate over these contexts:

$$l(v,c) = \arg\max_{l'} \sum_{(u,c') \in N(v,c)} \delta(l(u,c'),l') \tag{3}$$

where c and c' are the contexts associated with the nodes and edges. The algorithm ensures that labels are propagated in a manner that respects the contextual dependencies between nodes.

Partition Quality Metrics

The effectiveness of the partitions generated by CARGO is evaluated using three key metrics: cohesion, coupling, and transactional purity.

Cohesion

Cohesion measures how strongly the nodes within the same partition are connected. Let P_i denote a partition, and $E_{\text{int}}(P_i)$ be the set of edges within partition P_i. The cohesion of a partition P_i is defined as:

$$\text{Cohesion}(P_i) = \frac{|E_{\text{int}}(P_i)|}{|E_{\text{int}}(P_i)| + |E_{\text{ext}}(P_i)|} \tag{4}$$

where $E_{\text{ext}}(P_i)$ represents the edges connecting nodes in P_i to nodes in other partitions. A higher cohesion value indicates a more tightly coupled partition.

Coupling

Coupling measures the degree of inter-dependence between different partitions. Let $E_{\text{cross}}(P_i,P_j)$ denote the set of edges between partition P_i and partition P_j. The total coupling between partitions is given by:

$$\text{Coupling}(P) = \frac{\sum_{i \neq j} |E_{\text{cross}}(P_i,P_j)|}{|E|} \tag{5}$$

A lower coupling value indicates that the partitions are more independent, which is desirable for microservices architectures.

Transactional Purity

Transactional purity measures the extent to which each partition interacts with a single database. Let D_k represent a database table, and let P_i be a partition that accesses D_k. The transactional purity of a partition is defined as:

$$\text{Transactional Purity}(P_i) = 1 - H(\{P_i(D_k)\}) \tag{6}$$

where *H* is the entropy function measuring the diversity of database access across partitions. A purity value of 1.0 indicates that a database is accessed by a single partition, minimizing the need for distributed transactions.

After the initial partitioning, CARGO iteratively refines the partitions by reapplying the context-sensitive label propagation over the SDG, ensuring that transactional purity is maximized, coupling is minimized, and cohesion is enhanced. This iterative process continues until the partitioning converges or achieves a predefined threshold for the quality metrics.

Comparative Evaluation and Results

In this section, we provide a detailed comparative evaluation of CARGO against other leading microservice partitioning tools, including Mono2Micro and CoGCN. The evaluation focuses on key architectural and system performance metrics such as cohesion, coupling, transaction purity, latency, resource efficiency, scalability, adaptability, and throughput. The robustness of these systems is further analyzed by examining failure rates and response times under various workloads. The evaluation was conducted using two benchmark Java applications, Daytrader and JPetStore, both of which represent complex enterprise-level systems.

Cohesion, Coupling, and Transaction Purity

One of the critical factors in effective microservice partitioning is achieving high cohesion within each service while minimizing coupling between services. CARGO consistently outperforms both Mono2Micro and CoGCN in these metrics, as seen in Figure 2. Higher cohesion means that CARGO produces partitions where components within the same service are more functionally related, leading to better maintainability and service independence. The lower coupling scores achieved by CARGO reflect its ability to reduce the dependencies between services, which is crucial for minimizing communication overhead and ensuring that services can scale and evolve independently.

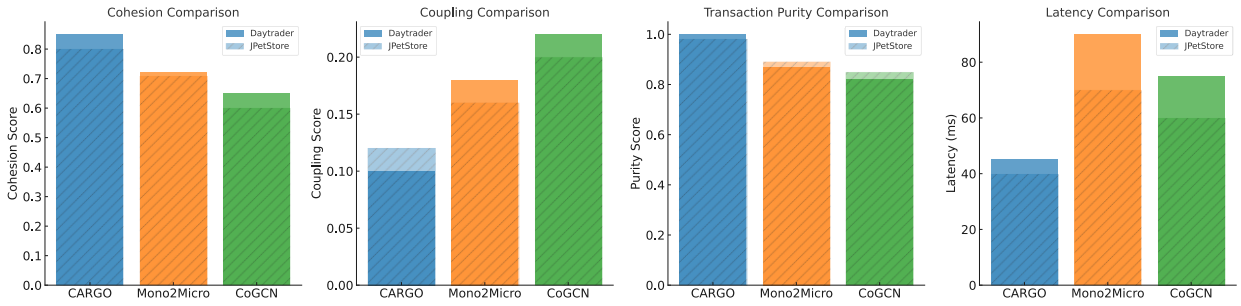

Figure 2 Comparative performance of CARGO, Mono2Micro, and CoGCN partitioning tools on key architectural metrics: Cohesion, Coupling, Transaction Purity, and Latency. CARGO consistently achieves higher cohesion, lower coupling, better transaction purity, and reduced latency for both Daytrader and JPetStore applications, demonstrating its effectiveness in creating more modular and efficient microservice architectures.

In addition to cohesion and coupling, transaction purity is a key metric that evaluates the extent to which partitioning avoids the use of distributed transactions. Distributed transactions can be costly in terms of performance and complexity. CARGO demonstrates superior transaction purity, with scores significantly higher than Mono2Micro and CoGCN, as shown in Figure 2. The framework ensures that the majority of transactions remain within a single service, thus avoiding the pitfalls associated with distributed transaction management, such as performance degradation and increased fault-tolerance challenges. Latency is another critical metric,

especially in distributed microservice environments where communication overhead between services can significantly impact response times. CARGO shows a clear advantage over Mono2Micro and CoGCN in reducing latency, as seen in Figure 2. The improvements in partitioning reduce inter-service communication, which directly contributes to lower response times. In real-world applications like Daytrader and JPetStore, the reduction in latency translates to faster transaction processing and a better end-user experience. Throughput, or the number of requests a system can handle per second, is another important performance indicator. CARGO significantly boosts throughput compared to the other tools, particularly in the Daytrader application (as seen in Figure 3). By optimizing the partition boundaries and reducing the need for services to frequently interact, CARGO allows the system to handle a higher volume of requests without bottlenecks. This makes CARGO particularly suitable for applications that require high concurrency and need to scale efficiently under load.

Resource Efficiency, Scalability, and Adaptability

Effective resource utilization is essential for maintaining system performance, especially as the number of microservices increases. CARGO demonstrates better resource efficiency across both applications (Daytrader and JPetStore), as shown in Figure 3. This results from the lower inter-service communication and more efficient transaction management, which reduce the overhead typically associated with microservice architectures.

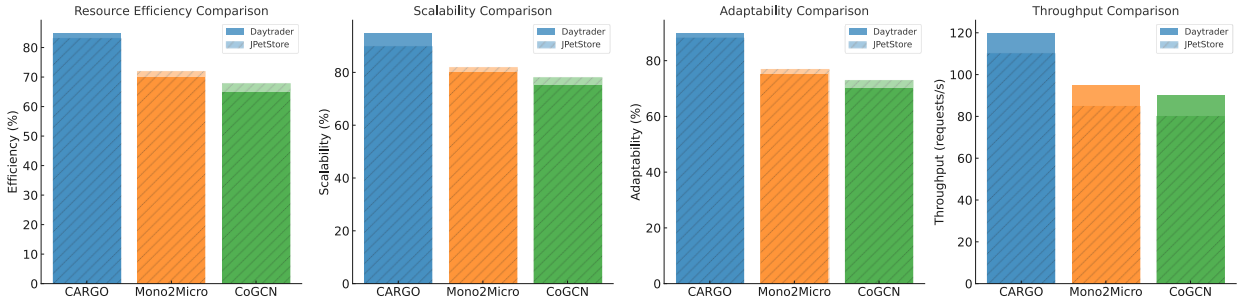

Figure 3 Comparative performance on Resource Efficiency, Scalability, Adaptability, and Throughput for Daytrader and JPetStore. CARGO shows superior resource utilization, better scalability under load, higher adaptability, and a significant throughput advantage, especially in the Daytrader application.

In terms of scalability, CARGO's partitioning scheme allows for better performance under high load conditions. The scalable nature of CARGO's architecture ensures that as the system grows and the number of transactions increases, the system can handle additional loads without a corresponding increase in response time or failure rate. Adaptability, or the system's ability to adjust to changing workloads and operational conditions, is also improved with CARGO. The context-sensitive partitioning approach ensures that the microservice boundaries remain robust and effective even as the system dynamically adjusts to different usage patterns, as seen in Figure 3.

Failure Rate and Response Time

Figure 4 provides a comprehensive analysis of failure rates and response times for Daytrader and JPetStore under varying loads. CARGO consistently maintains lower failure rates compared to Mono2Micro and CoGCN, especially under high load conditions. This indicates that CARGO partitions are more robust and less prone to failure when the system is subjected to stress. Furthermore, response times for CARGO remain competitive, even as the system handles

increasing transaction volumes. This shows that CARGO's microservice partitions are more resilient and capable of sustaining high performance under pressure.

The combination of lower failure rates and reduced response times is critical for systems that require high availability and reliability, such as financial trading platforms and e-commerce systems. In the Daytrader application, for example, CARGO manages to keep the failure rate below 0.04%, even as response times hover between 50ms and 65ms. Similarly, in JPetStore, failure rates remain low, and response times stay within an optimal range, demonstrating the effectiveness of CARGO in handling real-world workloads.

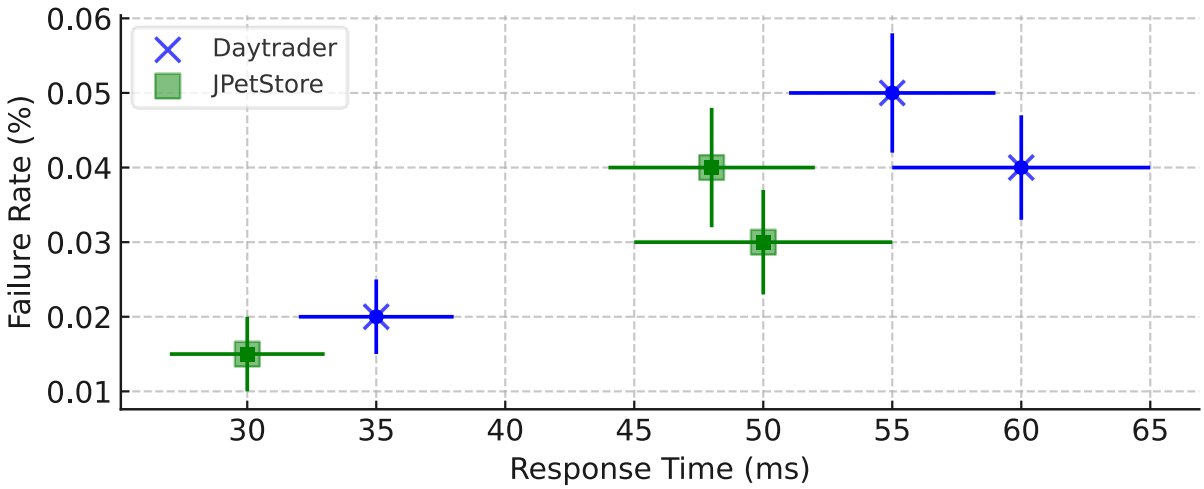

Figure 4 Failure rate versus response time for Daytrader and JPetStore applications under CARGO partitioning. CARGO maintains lower failure rates and competitive response times across a wide range of system loads, demonstrating robust performance and reliability in handling high transaction volumes.

Conclusion and Future Work

This paper introduced CARGO, an AI-driven framework leveraging System Dependency Graphs (SDGs) and a novel context-sensitive label propagation algorithm to enhance the partitioning of monolithic applications into microservices. Through comprehensive evaluations using benchmark applications such as Daytrader and JPetStore, CARGO demonstrated its superiority over existing tools like Mono2Micro and CoGCN across key metrics including cohesion, coupling, transaction purity, latency, and throughput. The framework successfully mitigates the creation of distributed monoliths and reduces the reliance on costly distributed transactions, ensuring more modular, efficient, and scalable microservice architectures.

Looking ahead, several opportunities exist to extend this work. Future research could explore the application of CARGO to other programming environments, such as Python and .NET, where monolithic architectures are also prevalent. This would require adapting the SDG construction and context-sensitive label propagation mechanisms to suit the specific characteristics and runtime behaviors of these ecosystems. Additionally, integrating machine learning-based optimization techniques to further automate the partitioning process could enhance adaptability and performance, making the framework applicable to a broader range of enterprise applications. Lastly, investigating the integration of CARGO with cloud-native orchestration tools and frameworks could further streamline the deployment of refined microservice architectures in real-world, dynamic environments.

REFERENCE

[1] R. Alboqmi, S. Jahan, and R. F. Gamble, “Toward enabling self-protection in the service mesh of the microservice architecture,” in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, CA, USA, 2022.

[2] A. El Malki and U. Zdun, “Evaluation of API request bundling and its impact on performance of microservice architectures,” in *2021 IEEE International Conference on Services Computing (SCC)*, Chicago, IL, USA, 2021.

[3] M. Seedat, Q. Abbas, and N. Ahmad, “Systematic mapping of monolithic applications to microservices architecture,” *Research Square*, 25-Aug-2022.

[4] M. Driss, D. Hasan, W. Boulila, and J. Ahmad, “Microservices in IoT security: Current solutions, research challenges, and future directions,” *arXiv [cs.CR]*, 17-May-2021.

[5] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, “Microservices: architecture, container, and challenges,” in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Macau, China, 2020.

[6] T. Cerny *et al.*, “On code analysis opportunities and challenges for enterprise systems and microservices,” *IEEE Access*, vol. 8, pp. 159449–159470, 2020.

[7] H. Noori, F. Mehdipour, M. S. Zamani, K. Inoue, and K. Murakami, “Handling control data flow graphs for a tightly coupled reconfigurable accelerator,” in *Embedded Software and Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 249–260.

[8] V. Arora, R. Kumar Bhatia, and M. Singh, “Evaluation of flow graph and dependence graphs for program representation,” *Int. J. Comput. Appl.*, vol. 56, no. 14, pp. 18–23, Oct. 2012.

[9] S. Sinha, M. J. Harrold, and G. Rothermel, “System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow,” in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, Los Angeles, CA, USA, 2003.

[10] V. Nitin, S. Asthana, B. Ray, and R. Krishna, “CARGO: AI-guided dependency analysis for migrating monolithic applications to microservices Architecture,” *arXiv [cs.SE]*, 24-Jul-2022.

[11] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, “Using service dependency graph to analyze and test microservices,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Tokyo, Japan, 2018.

[12] S. Luo *et al.*, “Characterizing microservice dependency and performance,” in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle WA USA, 2021.

[13] C. Yao, M. Zhang, Q. Lin, B. C. Ooi, and J. Xu, “Scaling distributed transaction processing and recovery based on dependency logging,” *VLDB J.*, vol. 27, no. 3, pp. 347–368, Jun. 2018.

[14] M. Sangeetha, J. R. Perinbam, and Revathy, “Hardware Estimation and Synthesis for a Codesign System,” in *2007 International Conference on Signal Processing, Communications and Networking*, Chennai, India, 2007.

[15] J.-P. Kim and J.-E. Hong, “A partition technique of UML-based software models for multi-processor embedded systems,” *KIPS Trans. PartD*, vol. 15D, no. 1, pp. 87–98, Feb. 2008.

[16] S. Malekshahi, M. Sedghi, and Z. Navabi, “Automating Hardware/Software partitioning using dependency Graph,” in *Proceedings of IEEE East-West Design & Test Symposium (EWDTS’08)*, Lviv, Ukraine, 2008.

[17] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, “A study of partitioning policies for graph analytics on large-scale distributed platforms,” *Proceedings VLDB Endowment*, vol. 12, no. 4, pp. 321–334, Dec. 2018.

[18] S. Liu, G. Tan, and T. Jaeger, “PtrSplit,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA, 2017.