Microservices Identification Using LLM

Jay Gandhi*,Raveendra Kumar Medicherla*,Manasi Patwardhan*,Dipesh Sharma†,Ravindra Naik‡
*TCS Research, Tata Consultancy Services, India †AMD, India ‡COEP Tech, Pune, India
Email:*gandhi.jay1,raveendra.kumar,manasi.patwardhan@tcs.com, †dipesh.sharma@amd.com, ‡ rdn.comp@coeptech.ac.in

***Abstract*—**
Identification of microservices within legacy monolithic applications is a critical, challenging, expert-driven task. Existing techniques often cluster technical elements of the application without aligning them with business domain. These technically inclined approaches may not fully address the larger legacy modernization objectives. To address this issue, we propose a novel approach that leverages Large Language Model (LLM) to infer the domain intent of key technical elements within the architecture. Our approach then combines these intents with architectural dependencies, and clusters them using Graph Neural Network (GNN) to identify candidate microservices that are domain-coherent. Preliminary evaluation across four benchmark applications of varying sizes and domains demonstrates the promising potential of our approach.

***Index Terms*—Microservices decomposition, Software Modernization, Graph Neural Networks**

I. Introduction

Several critical enterprise legacy applications across different domains are built on the *monolithic* architecture. These applications interleave business functions across different layers with technical concerns such as exception handling, transaction, and resource management. These applications are easy to deploy but challenging to scale and maintain, often leading to single points of failure [1].

To meet fast-changing organizational needs, ensure fault resilience under varying workloads, and enable effective utilization of cloud architectures, enterprise legacy applications need to be refactored into *microservices*. Each microservice can be independently instantiated and scaled to fluctuating workloads. They can be quickly re-instantiated in case of failure, ensuring system stability [1].

Refactoring a monolithic application into microservices architecture is a complex undertaking. A key and challenging first step is the identification of microservices [2]. In practice, an expert application architect carefully identifies microservices that align with business functionality by employing best patterns [3] and practices [4] from their *experience*. Such manual identification of candidate microservices is a time-consuming and expensive task.

A. Key Challenges

Recent techniques for identifying microservices mainly group together components that are technically *cohesive* [5]–[9]. However, they often overlook grouping operations that serve the same business purpose. This gap has been noted in a few studies [10]. Thus, candidate microservices recommended by such techniques may lack alignment with the underlying *business domain*, resulting in unrelated business operations being clustered together. Also, the *number* of microservices identified by the automated techniques is much higher than the number of microservices identified by the expert architects [11]. The *sizes* of the candidate microservices are highly skewed; there are a few large-sized candidates ("*God partitions*") and a large number of small-sized candidates [11].

Our aim is to identify candidate microservices that are aligned with domain services, while balancing both technical and domain dependencies. A *domain model* describes a *business domain* expressed in a ubiquitous language that is understandable to analysts. It contains domain concepts as vocabulary, a set of services and their operations, processes, rules, and product features. The industry-standard domain models for several industries are readily available [12], or an enterprise may develop their model using *event storming* [3].

B. New Ideas and Possible Explorations

The key idea is that the microservices identification process should first *align* key *architecture elements*, such as variables/fields and methods/functions of the application, with the domain vocabulary and operations. Then aggregate the cohesive *domain-aligned* architecture elements through clustering techniques to identify the candidate microservices.

The key challenge in *domain alignment* is to *map* the architecture elements onto semi-formal and textually expressed domain vocabulary and service operations. To address this challenge, instead of creating the mapping, we leverage program summary generation through suitable prompting of Large Language Models (LLMs). Our approach creates the *domain intent* of architecture elements rather than their technical intent. To understand the difference, consider a Java method in column A of Table I from the *pet clinic* application. The LLM generated domain intent and technical summary for this method are shown in columns B and C respectively. A domain analyst can more easily comprehend domain intent in column B than the technical summary in column C.

Our approach, using LLMs, creates domain intents of the architecture elements and transforms them into an *embedding* space. The approach utilizes only part of the domain model that describes domain vocabulary and service operations. It creates a weighted graph with the embeddings as nodes and relations between architecture elements as edges. Using a Graph Neural Network (GNN) based clustering algorithm, it identifies the cohesive nodes that form the candidate microservices. During the clustering process, it uses various

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00013

architectural metrics to evaluate the proposed clusters and recommends the most optimal clusters.

TABLE I: Domain and Technical Summaries of a Java Method

Method Source Code (A)	Domain Intent (B)	Technical Summary (C)
`Owner findOwner(int ownerId){ return ownerId == null ? new Owner(): this.owners. findById(ownerId);}`	Retrieves or creates an Owner entity based on the provided owner ID for pet registration and management purposes.	Method fetches an Owner entity by ID, returning a new instance if ID is null or not found.

II. Related Work

An extensive survey of microservices identification is available in [10]. Most techniques construct a graph from monolithic application using either architectural recovery [7], [8], [13], or program system dependency graphs [6], [7], or dynamic analysis [9], [14]. All these techniques use different clustering techniques and further *refine* the candidate microservices using optimization techniques [8]. A few techniques used domain elements such as entities, use cases, and services during microservices identification [8], [12], [15]. However, these techniques use heuristics and require extensive manual text extraction and annotation of code.

Graph Neural Network(GNN) based techniques construct graphs with some program features and their relations as nodes and edges. Node information is propagated to neighboring nodes, and the network is trained on specific objectives, such as modularization. Techniques differ in the graph creation and objective on which the network is trained. Chen D.*et al.* [7] used classes as nodes and control/data flow dependencies as edges. Clustering is performed by aggregating k-hop neighbor features. COGCN++ [16] uses normalized access patterns to other programs and resources and uses *k*-means++ as clustering algorithm. These techniques do not use domain information. Trabelsi et al. [5] used application call graph and used word embeddings of method names. They use Deep Modularity networks (DMoN) architecture [17] for clustering.

III. Approach

Our approach of domain-aligned microservices identification is shown in Figure 1. It has three major steps: Architecture Recovery, Domain Alignment and Service Identification.

A. Architectural Recovery

This step uses a static source code analyzer to extract architecture *elements* and various *relationships* between them from the monolithic application. An architecture element can be a data element (field) or an operation (method) that is a member of a program (or class). We identify six types of relationships, such as method call, class member, parameter, inheritance, composition, and aggregation. This step utilizes user-provided naming patterns to include code that implements business logic, dropping system, library, utility, or test code.

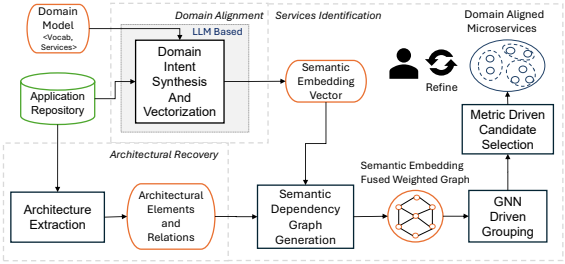

Fig. 1: Domain Aligned Microservices Identification

B. Domain Alignment

Domain Intent Synthesis: For each identified architecture element, the domain intent is synthesized by systematically prompting the LLM using a zero-shot prompting technique with the domain vocabulary, domain operations, field types, names, and method bodies including comments.

Prompt Design: The prompt has two sections. The *Overview prompt* section with the following parts, set up the overall task:

Role Description: It specifies the role, with a brief description about application and domain in which it operates and defines terminologies used in the rest of the prompt.
Task Description: It describes the high-level task of domain intent generation.
Summary Instructions: It consists of a set of instructions on the kind of intent that LLM expected to generate.

The *Specific prompt* section with following parts contains actionable steps to generate the intents.

Domain Description: This describes relevant domain vocabulary, services, and their description;
Package and Class: This is the chosen package and class within which the selected method or field is present;
Method or Fields: This part contains class level fields or method source code including comments, if available.
Domain Intent Query: Query instructing the LLM to generate a domain-aligned intent.

Domain Intent Vectorization: This step encodes the synthesized domain intents into vector embeddings, using a transformer-based sentence embedding model [18]. These embeddings form the input feature representations for the Graph Neural Network (GNN). They capture semantic meaning of the intents being encoded.

C. Services Identification

This step uses the architecture model and encoded domain intents to identify candidate microservices.

Semantic Dependency Graph Generation:

We create a weighted graph with embedding vectors as the node set, and dependency relationships among the architecture elements as edges. Predefined weights, which can be either positive or negative, are assigned to each edge depending on the relationship type. The overall weight of an edge is determined by aggregating the weights assigned to it.

GNN-driven grouping: To partition the semantic dependency graph and create clusters, we utilize the Deep Modularity Networks (DMoN) architecture. It is an unsupervised graph pooling method that optimizes clustering through modularity maximization. It supports end-to-end differentiable training, allowing the model to learn meaningful cluster assignments while preserving both local and global graph structures. Its ability to perform clustering without requiring labeled data, combined with its emphasis on structure-aware node aggregation, aligns well with our decomposition objective. A set of clusters identified by a GNN is referred as a *decomposition*. However, DMoN-based decomposition requires the size of decomposition (k) as a parameter, and k is not known *a priori*. To address this, we run GNN decomposition for values of ranging from 2 to $N/2$, where N represents the number of classes considered.

Metric-driven candidate selection: This step evaluates the identified decomposition to select the most suitable candidate microservices. Each decomposition is assessed against a set of well-defined architectural quality metrics such as modularity [19] and cohesion [19]. The decomposition that performs best is suggested as the candidate domain-aligned microservices. An expert architect can iteratively refine the candidate microservices based on their expertise.

IV. Implementation and Evaluation

A. Implementation

We have implemented our *GNN and Embedding-based Microservices Identification System* (GEMS) as a tool targeting legacy Java applications. The tool uses WALA(v1.6.0) [20] to recover the architecture. We have used *Llama 3.1-8B* as LLM to create domain summaries and *all-mpnet-base-v2* [18] sentence transformer model for generating the vector embeddings of the domain summaries. We have reused graph creation components of Service Cutter [15] to generate a weighted graph and implemented metrics evaluation using Python.

B. Evaluation Setup

Benchmarks: Table II shows an overview of the size of the chosen open-source benchmarks along with their respective domain and number of domain services used in our experimentation. These representative enterprise applications have been used in earlier evaluations [5], [6], [11]. For all these applications, corresponding domain models are crafted manually with the help of GPT-4o [21] LLM.

TABLE II: Benchmark applications and their domain services

Benchmark	#Classes	#Methods	#LOC	Application Domain	#Domain Services
JForum 3 [22]	352	2376	32221	Online Forum	14
Daytrader 7 [23]	109	952	14372	Stock Trading	11
FXML-POS [24]	55	426	4720	Point of Sale	8
PetClinic [25]	23	84	1334	Veterinary Clinic	4

Evaluation methodology: We evaluated GEMS and compared its performance against two state-of-the-art techniques: CARGO (CAR) [6] and MAGNET (MAG) [5]. CARGO clusters system dependency graph. MAGNET uses call graph based GNN clustering.

TABLE III: GEMS Evaluation

Benchmark	Tool	Cov	Partition Metrics				Quality Metrics				
			Count	Size			CHM	CHD	SMQ	CMQ	MojoFM
				Avg	Max	Min					
JForum 3	CAR	38%	174	3.6	69	1	0.63	0.61	0.054	-1.3e-4	62.7%
	MAG	79%	70	18.4	513	1	0.62	0.49	0.04	-3e-4	51.8%
	GEMS	82%	133	11.5	64	1	0.217	0.55	0.31	2.3e-2	61%
Daytrader 7	CAR	39%	72	5	42	1	0.48	0.72	0.02	-3.1e-5	71.6%
	MAG	63%	36	16.1	48	1	0.43	0.66	0.03	-7e-4	59.18%
	GEMS	83%	43	17.76	33	1	0 .18	0.72	0.33	1.6e-2	36.5%
FXML-POS	CAR	18%	15	4.9	39	1	0.59	0.68	0.067	4e-6	89.2%
	MAG	79%	17	21.9	90	1	0.6	0.57	0.02	-9e-4	60.2%
	GEMS	84%	26	14.84	19	9	0.27	0.617	0.39	5.7e-2	49.3%
PetClinic	CAR	21%	10	2.3	4	1	0.42	0.70	0.05	8.2e-5	77.1%
	MAG	69%	14	5.5	14	1	0.51	0.59	0.06	-2e-32	40.26%
	GEMS	87%	10	9.7	17	4	0.046	0.66	0.34	2.2e-2	29.7%

Cov: Coverage,%-age of application methods considered for clustering.

Evaluation Metrics: To evaluate the *quality* of the identified microservices, we used five metrics. The *Cohesion at Message level* (CHM) [26] metric measures the cohesion using method calls, whereas *Cohesion at Domain level* (CHD) [26] metric measures the cohesion in terms of domain, based on method signatures. The *Structural Modularity Quality* (SMQ) [14] metric evaluates the structural modularity and *Conceptual Modularity Quality* (CMQ) [14] metric assesses the domain alignment. The fifth metric is an architectural *distance* metric, MojoFM [11] that represents architecture drift, hence the amount of *refactoring* required to transform monolithic architecture into microservices architecture. For the purpose of calculation of MojoFM, we considered each package of the monolithic application as a module.

Identified microservices should be highly cohesive, modular, and loosely coupled, without incurring excessive refactoring effort. Thus, candidate microservices with *higher* values of SMQ, CMQ, CHM, CHD and *lower* value of MojoFM are desirable. In addition to these metrics, we compute the partition metrics of microservices in terms of sizes of clusters where each cluster is a set of architectural elements. We also measure *coverage*, which represents the proportion of methods in the monolithic source code that are present in the identified microservices architecture.

C. Results

Table III shows partition size statistics, quality metrics, and coverage for all the tools. Values indicating the lowest coverage in *Coverage* column and the largest partition size (god cluster) in *Max Size* column are highlighted in red colour. Values highlighted in blue colour in rest of the columns represent metrics where tools other than GEMS performed better.

Among the evaluated tools, GEMS exhibits higher coverage, suggesting a broader inclusion of application components during clustering. CAR consistently exhibited low coverage. Notably, GEMS consistently generates uniform size clusters, avoiding god clusters. In contrast, MAG and CAR exhibit the presence of imbalanced clusters.

The number of partitions alone is not sufficient to assess clustering quality. However, when coverage and the size of

the largest partition metrics are combined, GEMS outperforms others by ensuring high coverage and balanced partition sizes.

In the case of MojoFM, GEMS consistently performed better than CAR and MAG, except for JForum, where MAG surpassed GEMS. The god cluster in case of MAG suggested that decomposition is influencing this. CMQ values indicate that GEMS outperformed all approaches by producing conceptually modular clusters. Improvement with CMQ highlights the effect of domain alignment provided by GEMS. In contrast, GEMS achieved competitive SMQ values without the presence of such skewed partitions. CHD does not demonstrate any significant improvement compared to other techniques. GEMS consistently produced clusters with low CHM. Further investigation is necessary to evaluate the underlying causes.

V. Discussion and Conclusion

a) Key Observations: This work demonstrates that LLMs effectively translate the architecture elements into domain intents by leveraging the domain model as context. This capability extends beyond service identification and can support tasks such as detecting semantically duplicate functions across applications. Additionally, LLMs can generate meaningful *service names* for candidate microservices.

Candidate services: The results presented in Table III highlight that the GEMS approach is promising with balanced performance by achieving high coverage, low MojoFM and good quality metrics. However, we observed that there are a few shortcomings across all approaches. Evaluating the candidates with isolated metrics may lead to incorrect conclusions. For instance, while CAR produced better CHM and CHD values in three out of four cases, it has lowest coverage due to omitted architectural components that are essential for business functionality. Second, manual review revealed that candidates occasionally deviate from the intended domain services. Third, the number of identified candidate microservices is still high. This shows that there is a need for post-processing of candidates. Finally, the current evaluation methods fall short of measuring the domain alignment.

b) Future directions: Future work will focus on enriching architecture relations with domain model elements derived from processes and use cases. Additionally, we propose clustering the service names of the candidate microservices to identify higher-level services as a post-process. We also plan to design a learning framework that captures expert's actions of merging or splitting the identified candidates as feedback to improve microservices identification.

A. Conclusion

This paper introduced a novel technique for identifying domain-aligned microservices in legacy enterprise applications using LLMs, sentence embeddings, and GNN. The proposed GEMS tool integrates LLM-generated domain intents into a relation-rich graph and applies GNN-based clustering with metric-driven evaluation. Experimental results on four benchmark applications demonstrate the effectiveness of this approach. These findings highlight the potential of combining LLMs and GNNs for microservices identification.

References

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[2] J. Bogner *et al.*, "Assuring the evolvability of microservices: insights into industry practices and challenges," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.

[3] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[4] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[5] I. Trabelsi *et al.*, "Magnet: Method-based approach using graph neural network for microservices identification," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 2024.

[6] V. Nitin *et al.*, "Cargo: ai-guided dependency analysis for migrating monolithic applications to microservices architecture," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[7] D. Chen *et al.*, "Enhancing microservice migration transformation from monoliths with graph neural networks," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025.

[8] G. Filippone *et al.*, "From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. IEEE, 2023.

[9] A. K. Kalia *et al.*, "Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices," ser. ESEC/FSE 2020. ACM, 2021.

[10] Y. Abgaz *et al.*, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, 2023.

[11] Y. Wang *et al.*, "Microservice decomposition techniques: An independent tool comparison," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE'24),*, 2024.

[12] J. Gandhi *et al.*, "Domain aligned microservices decomposition," in *Proceedings of the 18th Innovations in Software Engineering Conference*, 2025.

[13] Y. Zhang *et al.*, "Software architecture recovery with information fusion," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[14] W. Jin *et al.*, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, 2019.

[15] M. Gysel *et al.*, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*. Springer, 2016.

[16] U. Desai *et al.*, "Graph neural network to dilute outliers for refactoring monolith application," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 1, 2021.

[17] A. Tsitsulin *et al.*, "Graph clustering with graph neural networks," *Journal of Machine Learning Research*, vol. 24, no. 127, 2023.

[18] K. Song *et al.*, "Mpnet: Masked and permuted pre-training for language understanding," *Advances in neural information processing systems*, vol. 33, 2020.

[19] T. I. Mohottige *et al.*, "Microservices-based software systems reengineering: State-of-the-art and future directions," *arXiv preprint arXiv:2407.13915*, 2024.

[20] "Wala." [Online]. Available: https://github.com/wala/WALA

[21] "Gpt4o." [Online]. Available: https://openai.com/index/hello-gpt-4o/

[22] "Jforum3." [Online]. Available: https://github.com/rafaelsteil/jforum3

[23] "daytrader7." [Online]. Available: https://github.com/WASdev/sample.daytrader7

[24] "Fxml-pos." [Online]. Available: https://github.com/sadatrafsanjani/JavaFX-Point-of-Sales

[25] "Petclinic." [Online]. Available: https://github.com/spring-projects/spring-petclinic

[26] D. Athanasopoulos *et al.*, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, 2014.