https://doi.org/10.51483/IJAIML.6.1.2026.150-167 ISSN: 2789-2557

SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

An AI-Assisted Approach for Microservice Identification in Service-Oriented Architectures Using Large Language Models

Sandeep Sharma[1*], Vijay Pal Singh[2]

[1]Department, of Computer Engineering & Applications, Mangalayatan University, Aligarh, India
[2]Department, of Computer Engineering & Applications, Mangalayatan University, Aligarh, India
E-mail: [1]20230102_sandeep@mangalayatan.edu.in, [2]drvijaypaltilotiya@gmail.com

Article Info

Volume 6, Issue 1, January 2026
Received : 15 September 2025
Accepted : 11 January 2026
Published : 26 January 2026
doi: 10.51483/IJAIML.6.1.2026.150-167

Abstract

Identification of microservices presents multiple challenges within legacy Service-Oriented Architecture (SOA) such as tightly coupled services, implicit domain logic, and insufficient tools for automated service boundary identification. This paper introduces an AI-assisted conceptual and methodological approach to demonstrate the transition from SOA to Microservices Architecture (MSA) during the initial phases within clear design boundaries and human oversight. Architectural Reasoning Context (ARC) is the core of this approach, a machine-readable representation that combines expert-enriched metadata with expert-defined service boundary rules. The ARC guides large language models (LLMs) by providing structured prompts to generate candidate microservice groupings. This approach, executed sequentially, starts with the analysis of source code and extraction of functional units, which are annotated by domain experts. The service boundary rules are formally encoded and combined with functional unit metadata to construct the ARC representation. Using LLM-based reasoning, the proposed method processes ARC to suggest candidate microservices in JSON format, which are iteratively refined via human-in-the-loop validation, and based on the finalised JSON groupings, it produces a reviewable report for expert evaluation. After validation, to support downstream migration efforts, a final microservice identification proposal is generated. This pipeline maintains architectural integrity, emphasises explainability, policy-compliant microservice identification and expert validation, while minimising manual effort. achieving an average Net Accuracy of 66.66–70.37% and Rule Compliance of 73.80–76.66%, with consistent alignment with expert decisions, demonstrating its effectiveness.

Keywords: Microservices, Microservice identification, SOA-to-MSA Migration, Large Language Models (LLMs), Architectural Reasoning Context (ARC)

INTRODUCTION

Service-Oriented Architecture (SOA) is widely used in enterprise systems because of its modular and reusable structure. However, changing software needs in industry have shifted focus towards Microservice Architecture (MSA). MSA provides better scalability, flexible deployment, and enhanced fault isolation. Regardless of these

advantages, transitioning from SOA to microservices is difficult because of the strong dependencies between legacy services, implicit domain logic, and the absence of reliable tools for systematic microservice boundary identification.

A key challenge in this transition is identifying accurate service boundaries that are suitable for microservice deployment. Traditional methods often depend on manual analysis, domain-driven design, or static code review. These methods are time-consuming, subjective, and may not scale effectively for large and complex systems. Moreover, they lack support for policy-driven decision-making and iterative refinement guided by expert-defined service boundary rules.

Despite efforts to address these issues, several unresolved barriers limit the effective application of existing methods to enterprise migration scenarios. In particular, most decomposition methods rely on structural relationships or dependency graphs to define microservices, which fail to consider semantic relationships or domain constraints necessary for accurately defining the boundaries of those microservices. Secondly, many of the clustering and embedding-based AI decomposition methods produce results that are opaque and challenging for architectural teams to validate or justify the microservices boundaries in enterprise systems. Thirdly, architectural policies are frequently neither established nor enforced during the automated decomposition of the architecture, leading to inconsistencies between the generated microservices and the organisation's design principles. Finally, existing approaches do not support reproducibility, due to the unstructured nature of input or the randomness associated with data-driven approaches.

Recent advancements in Large Language Models (LLMs) have demonstrated their potential in analysing and generating software artefacts. However, applying them to different software architecture tasks, such as the identification of microservices, requires structured inputs and context-sensitive guidance. Largely dependent on embeddings or unstructured prompts, current LLM-based approaches lack explicit architectural or domain-specific guidance. As a result, LLM-based methods produce opaque and non-deterministic decompositions that are difficult to validate when used in enterprise settings. This study proposes a structured and explainable AI-assisted approach to overcome the above issues.

The core of this approach is the Architectural Reasoning Context (ARC), a machine-readable JSON artefact that synthesises functional-unit metadata with expert-defined service boundary rules to guide LLM-based reasoning. The presented approach is validated on two domain-representative SOA systems, achieving an average Net Accuracy of 66.66–70.37% and Rule Compliance of 73.80–76.66%. The results demonstrate reduced manual effort while maintaining effectiveness, explainability, and policy-compliant microservice identification.

The main contributions of this paper are:

1. A novel Architectural Reasoning Context (ARC) approach that allows the use of structured input and the integration of architectural rules for microservice decomposition.
2. A dual-agent LLM-based pipeline that generates microservice groupings and human-readable documentation.
3. A validation-driven mechanism for ensuring consistency, correctness, and reproducibility of outputs.
4. An empirical evaluation was conducted across two domains to demonstrate the effectiveness of the proposed approach for microservices identification.

This research contributes to bridging the gap between automated decomposition methods and practical architectural decision-making in real-world systems.

RELATED WORK

The transition from Service-Oriented Architecture (SOA) to Microservice Architecture (MSA) has gained significance in academia and industry primarily because of MSA’s advantages in terms of scalability, maintainability, and deployment agility. Rule-based methods or domain-driven design (DDD) are early decomposition strategies to define service boundaries, focusing on service grouping and discovery strategies[1],[24] and migration practices [2]. In heterogeneous legacy systems, these approaches required immense manual effort and lacked adaptability.

To manage structural complexity, later research introduces taxonomies of decomposition patterns and identifies pitfalls in migration processes, providing useful practical insights [3],[4]. To improve the accuracy of service boundary identification, dataflow-based methods were developed, both static and dynamic, which were effective in inferring control dependencies and runtime interactions. Notably, control flow graphs and module interaction patterns were leveraged to identify candidate microservice clusters [5], [6].

To capture architectural patterns and enforce design guidelines in a structured way, model-driven engineering and transactional context modelling plays a significant role [7],[8],[9]. Industry case studies emphasise the necessity of iterative validation and the participation of specialists in practical service identification workflows [10],[11].

In recent years, AI-automated methods have gained momentum. In partially automating the service boundary identification process, unsupervised learning and clustering algorithms have shown strong performance with

minimal expert intervention [12],[13] and earlier approaches have also contributed to migration understanding [14]. Different techniques such as graph-based and scenario-driven approaches have improved understanding of service cohesion and domain boundaries [13].

However, these approaches have significant limitations. Graph-based algorithms like MAGNET [15] use Graph Neural Networks (GNNs) to analyse graph structure, but they lack the interpretability and fail to apply domain-specific architecture constraints. Systematic literature reviews [16],[17],[18], show that there have been advances automating solutions, but existing methods struggle with providing explanatory data and alignment with expert-determined architectural intent.

Building on these advancements, current research has started to investigate how LLMs help identify microservice boundaries within software architectures. However, there are only a few examples where LLMs are leveraged to perform this task. For example, MicroDec [19] and MonoEmbed [21] demonstrate that LLM embeddings are helpful in decomposing an application into microservices, but they primarily employ LLMs as representation generators, and do not exploit policy-based reasoning to make decisions. Similarly, MicroRec [20] leverages LLMs for microservice recommendation but provides limited support for incorporating explicit architectural constraints. Recent research includes comparative analysis of tools for decomposition techniques [22], systematic reviews on how LLMs help generate microservices [23], highlighting the increased interest in automating the decomposition of applications into microservices. However, the methods identified above do not incorporate any explicit means of including architectural constraints within them, which limits these methods' transparency and reproducibility.

Table 1 shows the evaluated approaches for identifying microservices using three different classes of microservice identification methods namely, traditional, machine learning, and LLM (Large Language Model) based reasoning. The evidence in this comparison shows that no method currently exists that integrates structured domain-specific knowledge with explicitly stated architectural rules, LLM-informed reasoning, and iterative expert validation within a unified reproducible pipeline. This gap motivates the present study.

TABLE 1. Comparison of microservice identification approaches across key dimensions

Dimension	Traditional (DDD, static)	ML/GNN (MAGNET, clustering)	LLM-embedding (MicroDec, MonoEmbed)	Proposed Approach
Input	Source code, manual analysis	Source code graphs	Source code + LLM embeddings	ARC (code metadata + expert rules)
LLM Utilisation	Not used	Not used	Embedding generator only	Structured reasoning with rule constraints
Explainability	Moderate(implicit rationale)	Low (opaque clusters)	Low (opaque clusters)	High (explicit rule traceability + justification)
Policy / Rules	Implicit	None	None	Yes (priority-based)
Reproducibility	Low	Moderate	Low	High (ARC versioning + fixed prompts)
Performance / Evaluation	Manual / qualitative	Moderate accuracy	Improved clustering	Quantitative (Net Accuracy + Rule Compliance)

METHODOLOGY

This study aims to leverage an AI-assisted approach for identifying candidate microservices from SOA-based systems using large language models (LLMs) and structured reasoning. Architectural Reasoning Context (ARC) is the core artefact of this method, which combines Expert-Enriched functional_units metadata and expert specified service boundary rules into ARC JSON (a machine-readable input). The ARC guides Microservice Identification AI-Agent in generating candidate microservice groupings. These JSON outputs are iteratively refined through Human-in-the-loop feedback, which results in a validated, policy-aligned recommended microservices proposal ready for migration execution. Figure 1 outlines the process the AI-assisted approach uses to recommend microservices when migrating from legacy SOA systems.

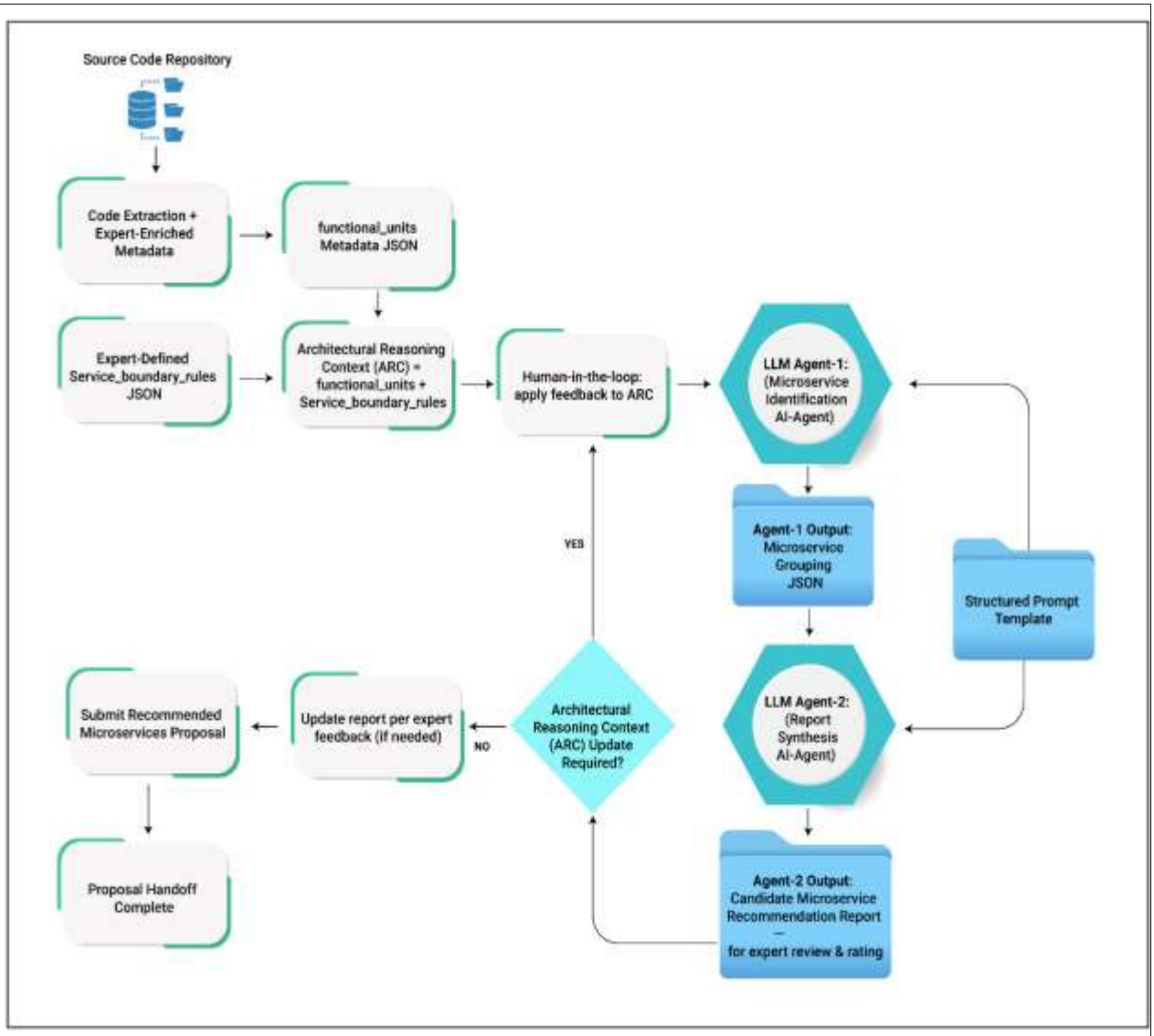

Fig 1. Methodology of an AI-Assisted approach for Candidate Microservice Identification

Algorithm: AI-Assisted Candidate Microservice Identification Using ARC and LLM Agents
Below is the step-by-step algorithm of proposed approach.

Algorithm 1 AI-Assisted Candidate Microservice Identification using ARC and LLM Agents

Input: Legacy SOA source code, expert knowledge - (service_boundary_rules.json)
Output: Final validated candidate microservice recommendation report

1: Collect legacy SOA source code, configurations, and interfaces
2: Create artefact package for static analysis
3: Perform static analysis to extract service classes, public methods, and inter-service dependencies
4: Produce Raw_Metadata
5: Construct functional_units.json from Raw_Metadata including: method, service, operation (CRUD), domain_hint, subdomain, service_dependencies, functional_summary
6: Enrich functional_units.json using expert input (refine domain_hint, assign subdomain, improve summary)
7: Produce updated functional_units.json
8: Define service_boundary_rules.json using expert-defined policies with fields: id, priority, description
9: Construct ARC.json by merging functional_units.json and service_boundary_rules.json
10: Review ARC.json for completeness and rule integrity
11: Produce approved ARC.json
12: Provide ARC.json, output schema, and structured prompt to Microservice Identification AI-Agent (Agent-1)
13: Agent-1 performs rule-aware grouping:
14: Group functional units by semantic cohesion (domain_hint, subdomain)
15: Refine groupings using service dependencies
16: Enforce service boundary rules in priority order (priority 1 = highest)
17: Generate candidate microservices $ms = (ms_1, ms_2, ..., ms_z)$
18: Output candidate_microservices_grouping.json containing:
19: microservice name, functional unit assignments, justification, satisfied rule IDs
20: Perform syntactic JSON validation on candidate_microservices_grouping.json
21: **if** JSON invalid **then**
22: attempt $\leftarrow$ 1
23: **while** JSON invalid AND attempt $\leq$ 2 **do**
24: Provide validator errors, schema, and previous output to repair prompt
25: Re-invoke Agent-1 to regenerate JSON
26: Re-validate JSON
27: attempt $\leftarrow$ attempt + 1
28: **end while**
29: **end if**
30: If JSON is still invalid then go to step 41
31: Perform schema validation against expected microservice schema
32: **if** schema invalid **then**
33: attempt $\leftarrow$ 1
34: **while** schema invalid AND attempt $\leq$ 2 **do**
35: Provide validator errors, schema, and previous output to repair prompt
36: Re-invoke Agent-1 to regenerate JSON
37: Re-validate schema
38: attempt $\leftarrow$ attempt + 1
39: **end while**
40: **end if**
41: **if** output remains invalid after maximum attempts **then**
42: Discard run and re-run after prompt or ARC refinement
43: **end if**
44: Provide candidate_microservices_grouping.json, output schema, and prompt to Report Synthesis AI-Agent (Agent-2)
45: Generate candidate microservice recommendation report (rpt_candidate_microservices) in markdown format
46: Present report to domain experts for review
47: **if** revisions required **then**
48: Update functional_units.json and/or service_boundary_rules.json or refine prompts
49: Reconstruct ARC.json
50: Repeat from Agent-1 execution step
51: **else**
52: Approve report
53: **end if**
54: Final validation of candidate microservice recommendation report by domain experts
55: Submit report to architecture and migration teams for final decision

Fig 2. Algorithm of an AI-Assisted approach for Candidate Microservice Identification

Preparation of Input for ARC Construction

The proposed methodology begins with the collection of artefacts and source code from legacy SOA-based systems, followed by enrichment through expert-annotated functional units and the definition of expert-driven service boundary rules. A hybrid strategy that combines static analysis with expert-driven augmentation is used to capture both explicit code structure and implicit architectural intent. The evolving artefact of this strategy is the Architectural Reasoning Context (ARC) serves as a structured input guide to the LLM for the service boundary identification process.

This structured preparation phase ensures that both technical dependencies and domain knowledge are explicitly encoded, improving the reliability and interpretability of downstream LLM reasoning.

Functional Unit Metadata

The initial step consists of extraction of semantic and structural metadata from the legacy codebase. To guide service boundary identification, metadata captures information about different modules such as service classes, public methods, and inter-service dependencies.

Different language-specific static analysers such as Roslyn (.NET) and JavaParser (Java) are used to identify class-level service definitions, public method signatures, and to support the construction of inter-module dependency graphs. The functional_units block within the ARC contains all these compiled outputs. Each unit includes:

1. ***method:*** Name of the publicly defined methods used to execute specific business processes.
2. ***service:*** Describes the service name where methods are written.
3. **operation:** Describe the type of operation the method performs (CREATE/READ/UPDATE/DELETE).
4. ***domain_hint:*** Initial domain keyword is practically obtained from method or API naming. For semantic alignment, experts review and adjust this field.
5. **subdomain:** Domain experts define canonical bounded context label (e.g., “Profile”, “Authorization”) based on underlying business logic and architectural intent.
6. ***service_dependencies:*** A list of other services that this functional unit depends on.
7. ***functional_summary:*** Experts update a short summary of the method's role, which is inferred from naming or comments.

Before being structured into ARC, these elements are enriched and annotated by experts to ensure semantic accuracy. The primary benefit of this enrichment phase is that it enhances the LLM's ability to generate accurate microservice groupings by integrating domain-based knowledge, which is missing in automated decomposition approaches.

To illustrate the JSON structure of a functional_units entry extracted from a legacy SOA system see Figure 3.

```
{
 "functional_units": [
  {
   "method": "<public method name from legacy SOA>",
   "service": "<legacy service name where the method exists>",
   "operation": "CREATE | READ | UPDATE | DELETE",
   "domain_hint": "<short domain keyword>",
   "subdomain": "<bounded context label>",
   "service_dependencies": ["<ServiceA>", "<ServiceB>"],
   "functional_summary": "<1–2 lines in business terms (expert-updated)>"
  }
 ]
}
```

Fig 3. Structure of functional_units JSON Object Extracted from a Legacy SOA System.

Extraction Heuristics for Functional Units

The initial step consists of extraction of semantic and structural metadata from the legacy codebase. To guide service boundary identification, metadata recognises different modules such as service classes, public methods, and inter-service dependencies.

A collection of heuristics is defined in a structured method as indicated in the previous section. This method follows a systematic approach across three analytical dimensions (i) selection of candidate functional units, (ii) interpret/assign semantics based on domain intent, (iii) identify structural dependencies and generate summary or overview. These heuristics also ensure transparency in how functional unit attributes are derived from the source code, creation. Thus, facilitating reproducibility of results across different systems.

Heuristics for Selecting Functional Units: Below are the heuristics used in functional unit selection.

1. ***Entry-point identification:*** Candidate methods are limited by the fact that only the publicly available methods associated with service entry points will be candidate methods; this includes those methods defined in Controller Classes, Service Interfaces, and any Externally Exposed Operations (such as WSDL or Open API). Therefore, non-functional artefacts like Framework Initialisation Code, Configuration Classes, DTO (Data Transfer Object) Mapper Classes and Utility Helper Classes will not qualify to remain in the pool of candidate methods.
2. ***Business operation filtering:*** A method is directly exposed as an operation by the service and called transitively via their call hierarchy. If the method contains business logic, it is included in the candidate methods. Examples of business logic include orchestration, validation, enforcement of rules. All methods that are only technical in nature (such as logging, string formatting, date handling, etc.) are filtered out and removed from consideration.
3. ***CRUD classification:*** Classification of each retained method utilises classification schemes such as CREATE, READ, UPDATE, and DELETE based on a composite heuristic comprising of the following three elements:
   - Lexical patterns of method name, e.g. "CREATE*", "GET*", "UPDATE*", "DELETE*".
   - HTTP Verbs associated with the annotated REST Endpoints, e.g. "GET" to the equivalent of "READ" and "POST" for "CREATE".
   - Semantic cues that indicate the nature of the action in the repository (or DAO - Data Access Object) invoked in the body of the method.

Heuristics for Deriving Domain Hints and Subdomains: Below are the heuristics used for deriving domain hints and subdomains.

1. ***Identifier tokenisation:*** The service names and methods are split based on various techniques (camelCase, underscores, etc.), for example: CreateUserProfile can be separated into individual tokens of "Create," "User," and "Profile." In addition, the common technical suffixes (e.g. Service, Manager, etc.) are eliminated to focus attention onto only those tokens that contain domain-specific information.
2. ***Domain vocabulary matching:*** Once the tokens have been generated, they are checked against a curated list of domain-specific vocabulary unique to the individual study. For example, the vocabulary of an e-Commerce domain would contain words like Cart, Order, and Payment, while the vocabulary of Infrastructure Event Management would contain words like Event, Alert, Incident and Notification.
3. ***Subdomain mapping:*** The domain_hint selected will be mapped onto the appropriate bounded context from the tasks defined (e.g., Order → Order Management).
4. **Expert validation:** Domain experts review and update any imprecise or generic domain_hints (for example, Process, Execute) to ensure that they accurately reflect the business context

***Heuristics for Identifying Service Dependencies*:** Service dependencies are identified using the following heuristics.

1. ***Static call graph construction:*** The inter-service dependencies for a given functional unit may be established as a result of static analysis of the method level call graphs. Such calls from one service class to another and vice versa become potential dependencies.
2. ***Aggregation at service level:*** Consolidating the granularity and complexity of the method level calls to “service level edges” (i.e., OrderService → PaymentService) retains the fundamental interaction patterns between services.
3. ***Filtering infrastructural cross-dependencies:*** Dependent services for cross-cutting resources (i.e., logging, emailing, and configuration) will not be included so that these services do not pollute the logic of the business services graph.
4. **Weak-coupling suppression:** Inter-service interaction that occurs infrequently, is used only for diagnostics, or are not the main function of the service (i.e., weak dependencies) will be excluded unless they affect the determination of microservice boundaries substantively.

Heuristics for Generating Functional Summaries: Functional summaries are generated using the following heuristics.

1. ***Template-driven summarisation:*** A functional_summary is created using a template formulation of the type operation method, domain hint and name tokens (e.g., "READ operation to request a customer's order history").
2. ***Leveraging inline documentation*****:** The informative comments or API annotations that are parsed are included to enhance the summary with the most natural descriptions, where available.
3. ***Expert refinement*****:** The domain expert is responsible for enhancing the accuracy of automatically created summaries, particularly in cases of nuanced logic (e.g., cart recalculation vs. cart update).

Service Boundary Rules

The Domain experts design a service boundary rules file that defines grouping policies and domain-specific architectural constraints in a structured JSON format to guide microservice identification. Strategic design preferences and technical boundaries are encoded to ensure organisational goals and domain logic-oriented microservices recommendation. Each rule entry in the JSON policy file typically includes:

1. ***id:*** A unique identifier for the rule (e.g., “R1”).
2. ***priority:*** An integer ranking that identifies the stronger preference among two or more rules to apply. where 1 denotes the highest priority
3. ***description:*** A textual explanation of the constraint or intent.

Service_boundary_rules block of the Architectural Reasoning Context (ARC) contains the expert-defined rules and constraints. These rules are utilised by the LLM during the generation of candidate microservices. The explicit encoding of the rules ensures policy compliance and traceable decision-making. Thus, addressing the lack of constraint-based decompositions in previous approaches.

All Service Boundary Rules in ARC express certain constraints on the attributes of one or more functional_units. Each rule represents constraints based on the service, operation, domain_hint, subdomain, and service_dependencies attributes discovered through source code analysis and expert annotation. The rules are provided through the pipeline to the LLM (Large Language Model) as structured, natural language conditions for constructing the prompts about those rules (e.g., functional_units with domain_hint=Payment must not be grouped together with Functional Units with domain_hint=Catalogue). Furthermore, when the model generates candidate microservices, it provides the identifier(s) of the rules referenced in the justification for the grouping decisions.

Precedence between rules is determined by their priority when there is a conflict. Higher-priority rules must be satisfied before lower-priority guidance is applied, and any violations of rules are recorded during human expert evaluation rather than discarding the entire candidate grouping. Consequently, lower-priority rules may be relaxed to accommodate higher-priority guidance and must be explicitly identified in the justification provided by the large language model and reviewed later by subject matter experts.

When provided with the ARC, the LLM acts as a rule-aware clustering assistant: it groups functional units primarily by semantic cohesion (domain_hint/subdomain and functional_summary), refines boundaries using service_dependencies evidence, and then enforces service boundary rules in ascending priority by splitting or reassigning units to avoid violations. It outputs a JSON candidate microservice grouping with justifications referencing rule identifiers, while dependency extraction and JSON/schema validation are handled externally before expert review.

To illustrate sample entries from the service_boundary_rules block in JSON format see Figure 4.

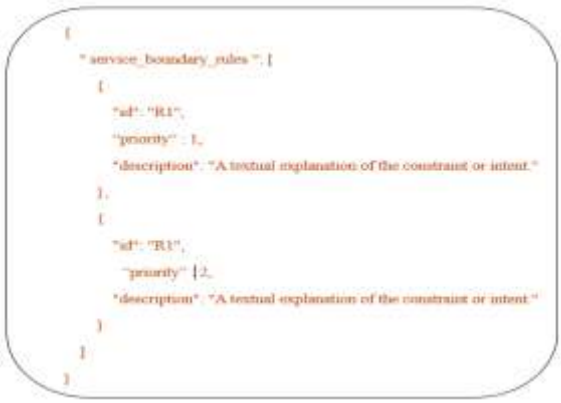

Fig 4. Sample Structure of the service_boundary_rules Block.

Architectural Reasoning Context (ARC)

The Architectural Reasoning Context (ARC) provides a core abstraction in the proposed approach. It is a structured, machine-readable JSON artefact that combines all inputs that are required for guided microservice identification. ARC serves as the bridge between raw code metadata and upper-level architectural logic to allow large language models (LLMs) to produce microservices recommendations that are technically consistent and policy-compliant. ARC acts as a unifying layer that integrates heterogeneous outputs into a unified, standardised format that allows for LLM-based reasoning.

In this approach, the LLM performs rule-aware grouping using the ARC as input, while ARC construction and output validation are handled externally to ensure correctness and reproducibility. The ARC integrates two primary inputs:

1. ***functional_units:*** This information, gathered from the source code, includes functional units with service/module names, public methods, dependencies, operations, and a short overview. Expert annotations like domain_hint and subdomain fields provide semantic grouping of the functional unit.
2. ***service_boundary_rules:*** Each rule carries an ID, priority, and description. The ARC enables LLMs to incorporate structural and contextual signals to reason about service boundaries and suggest candidate groupings that are refined through expert input.

Both the ARC representation of code structure and the rules defined by experts are necessary to capture two key aspects of the design of the system: the structure of the system and the intent of the architect. The service_boundary_rules, in particular, capture the strategic domain knowledge and the architectural principles and the organisational constraints that will be used to define the boundaries of the microservices. Using just policy rules for the definition of the microservice boundaries risks having architecturally sound microservices that are not technically viable. On the other hand, using the structural analysis for defining microservice boundaries may miss important domain semantics and governance requirements. Therefore, by bringing these two aspects together, ARC ensures that the microservices recommended by the LLM are aligned with the implementation, comply with policy, and are appropriate for real-world SOA-to-MSA migrations.

Additionally, the structured nature of ARC supports traceability and auditability of each microservice grouping as they are directly linked to specific functional units and rule identifiers.

Figure 5 illustrates two core blocks of Architectural Reasoning Context (ARC) : functional_units and service_boundary_rules.

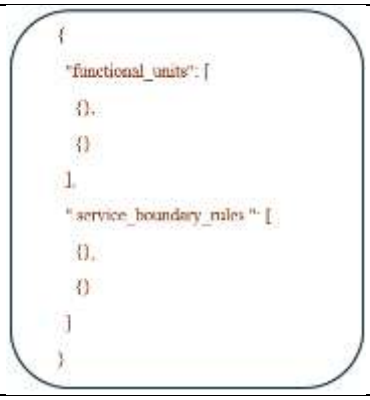

Fig 5. Primary Blocks of the Architectural Reasoning Context (ARC)

Structured Prompt Design

The proposed methodology utilises a dual-agent LLM (Large Language Model) pipeline, where Agent-1 (Microservice Identification AI-Agent) interacts with the Architectural Reasoning Context (ARC) to generate candidate microservice groupings in structured JSON format. Agent-2 (Report Synthesis AI-Agent) uses the structured JSON produced by Agent-1 to generate a human-readable summary report for expert review. The accuracy and clarity of the microservice recommendation are significantly influenced by the accuracy of the Architectural Reasoning Context (ARC) and the delivery of prompts. To keep things consistent, interpretable, and aligned with domain rules, the approach follows a standard prompting strategy guided by three core principles.

Role-Based Prompting

To align reasoning with software architecture practices, each agent is assigned a specific role. For instance:

"You are a senior software architect responsible for identifying candidate policy-compliant microservices from a legacy SOA system. Use the provided Architectural Reasoning Context (ARC), which contains functional units and service boundary rules, to guide your decisions. Group related methods based on their business responsibilities, structural dependencies, and the constraints defined in the rules. Generate the microservice groupings accordingly."
Persona-based prompts help to reduce irrelevant generalisations and improve contextual alignment with the intended task.

Chain-of-Thought Prompting for Reasoning Transparency

To enhance explainability, prompts guide the LLM to reason step-by-step by:

- Analysis of service responsibilities.
- Reference to architectural rules.
- Justification of each grouping decision.

This promotes transparency and supports human-in-the-loop validation.

Constrained Output Formatting

The prompts enforce structured output generation:

1. **Agent1(Microservice Identification AI-Agent)** generates a machine-readable JSON (Candidate_Microservices_grouping.json), including microservice ID, name, justifications, and satisfied rules ID. Figure 6 illustrates the structure of the JSON output produced by Microservice Identification AI-Agent (Agent 1).

Fig 6. Structure of the JSON Produced by Agent 1

2. ***Agent 2 (Report Synthesis AI-Agent)*** generates human-readable summary in markdown format for expert review, from the JSON output of Agent-1(Microservice Identification AI-Agent). Table 2 outlines the structure of the summary report generated by Agent 2(Report Synthesis AI-Agent) , including the microservice ID, microservice name, associated justifications, and the satisfied rule IDs for each grouping.

Table 2. Structure of the Summary Report Generated by Agent 2(Report Synthesis AI-Agent).

ID	Microservice Name	Justifications	Satisfied Rule ID
<id>	<microservice-name>	<Justification 1>	R#

		<Justification 2>	

Invocation Pipeline

The workflow uses a dual-agent invocation pipeline, which allows prompt execution using the Architectural Reasoning Context (ARC) and structured prompt templates to interface with LLM APIs.

1. ***Agent 1(Microservice Identification AI-Agent)*** accepts the Architectural Reasoning Context (ARC) and returns a structured JSON output proposing microservice groupings with justifications for consideration. Figure 7 illustrates the Prompt structure used by Agent 1(Microservice Identification AI-Agent) to derive microservice recommendation JSON from the ARC JSON input.

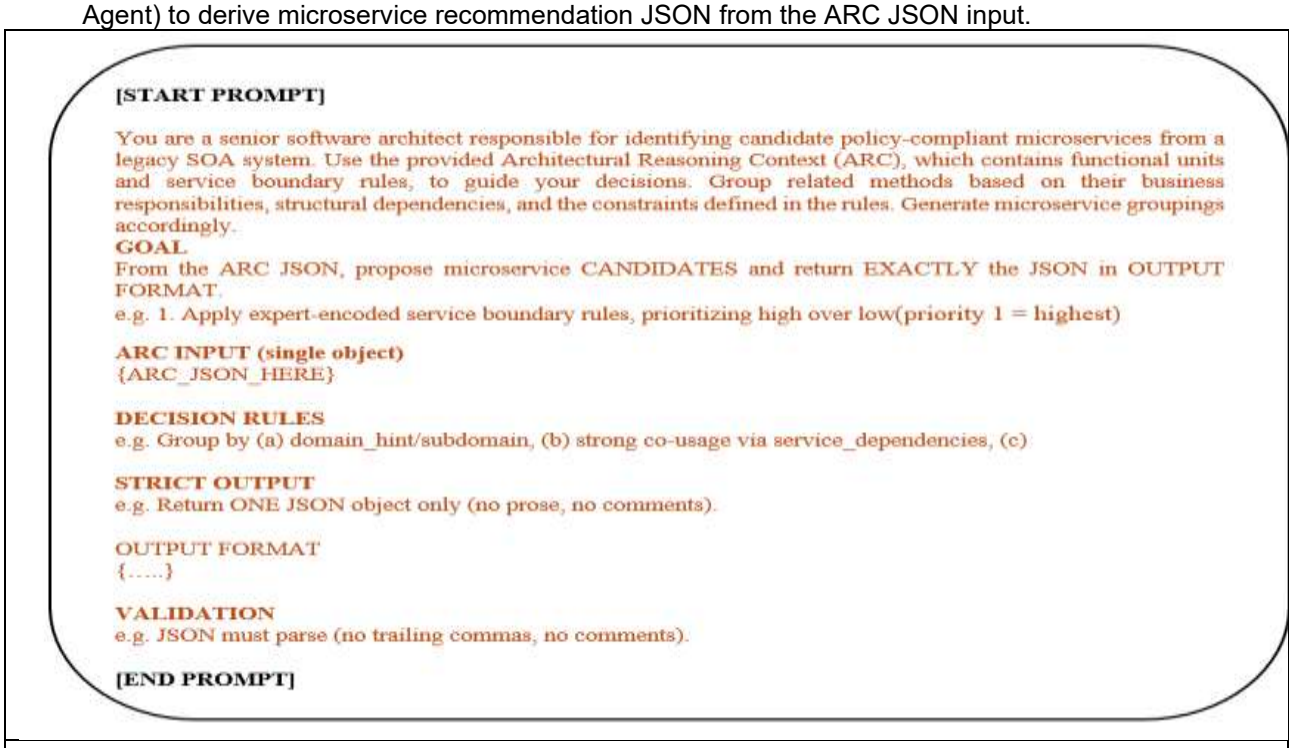

Fig 7. LLM Prompt Structured Used by Agent 1(Microservice Identification AI-Agent).

2. ***Agent 2 (Report Synthesis AI-Agent)*** consumes Agent-1's JSON output and returns a summary report in markdown format for expert review and validation. Figure 8 shows the Prompt structure used by Agent 2(Report Synthesis AI-Agent) to generate Microservice recommendations summary report based on the JSON input produced by Agent 1(Microservice Identification AI-Agent).

Fig 8. LLM Prompt Structured Used by Agent 2(Synthesis AI-Agent)

Prompts are executed with controlled parameters including:

- Set temperature (0.2–0.3) to reduce randomness.
- Max Tokens to control response length.
- Clear system role declaration and stop sequences to ensure format fidelity.

The outputs produced by the LLM agents (including microservice groupings {typically in JSON format} and recommendation reports {typically in markdown format}) are logged separately, versioned, and evaluated iteratively by experts for modification to the ARC when necessary.

The pipeline provides a way to provide repeatable and policy-aligned microservice boundary identification that is designed to be model-agnostic and can be integrated with LLMs that are either proprietary or open-source products. This is the operational centre of the proposed framework for scalable, explainable, and expert-verified microservice identification.

Error Handling, Fallback, and Output Validation

In order to improve robustness when interacting with LLMs and to reduce variance in output, the prototype uses explicit error-handling, fallback, and validation steps between ARC Construction and the two AI agents. Although implementing a complete system-level solution is outside the scope of this phase. The above mechanisms have been demonstrated and validated through a prototype simulation to evaluate the consistency and dependability of the core process flow.

The Microservice Identification AI-Agent (Agent-1) is validated through two phases when producing an output:

1. ***Syntactic JSON validation*****:** The model output, in terms of well-formed JSON, is validated in its raw form. If the output contains prose combined with JSON, it has commentary following the JSON output, or has an incomplete structure, the output is invalid.
2. ***Schema validation:*** if the output JSON is syntactically valid, it is then validated against the expected Candidate_Microservices_grouping.json schema. Failures to include elements such as microservice identifier and name, functional_unit assignment(s), reference rule identifiers as mandatory elements and types will lead to a non-conforming output.

When an Agent-1 (Microservice Identification AI-Agent) reply fails a syntactic or schema check, the same model is given a repair prompt, after which the model generates a new response. This prompt includes the last response, a brief account of why the previous response failed on one of the validation checks, and an instruction to recreate the JSON response and adhere to the original intended grouping of data, and the format of the JSON defined in the schema. Two attempts are allowed for the repair, but if the JSON is still invalid after these two attempts, the run is excluded from quantitative analysis and manually assessed.

Before invoking Agent-2 (a report synthesising AI-Agent), the JSON artefact that has passed validation is re-examined to ensure that the artefact is in accordance with the schema and the correct version. Because the output from Agent-2 is in report form rather than a JSON structure, only basic structure checks (i.e. checking that all of the required sections are included and not empty) are performed. When the report appears to be truncated or malformed as a result of the report generation process, the repair prompt will be issued in the same manner described for Agent-1.

In order to maintain traceability, each successfully validated JSON artefact is assigned a unique version (for example, through the use of a version identifier). As a result, downstream actions (Report Generation, Expert Review and Metric Calculation) will operate from an exact version of the ARC and Candidate Grouping(s); thereby providing assurance of the quality of the final proposal, with a means to reproduce it, roll back to earlier candidate grouping(s), and perform a complete audit on how the final microservice proposal(s) were built from previous ARC updates and generated outputs from LLMs.

By using this validation-first method, the evaluation metrics of Net Accuracy and Rule Compliance are applied only to artefacts that are of a sound structure, conform to the schema, and are reproducible. This applies to the constrained prototype simulation of this research project.

Human-in-the-Loop Validation

The LLM initially produces groupings of microservices based on the ARC, which are subsequently evaluated by experts to ensure architectural integrity and domain alignment. A review interface presents with the model's recommendations including, the methods comprising each grouping, associated rule annotations, and reasoning provided in natural language. Experts review these suggestions to ensure model's compliance with rules and modify the groups where necessary. Rule conflicts are resolved based on the rule priority: groupings are structured to satisfy higher-priority rules, and any remaining violations are corrected during expert review. Lower-priority rules may be relaxed only when they conflict with a higher-priority rule and must be explicitly justified and reviewed. All edits are versioned for transparency and traceability. Furthermore, observations from reviewers can also be included in future prompts for model behaviour refinement. The validation phase

balances automation with expert judgment to ensure decompositions are both technically valid and contextually sensible migration options for practical uses.

Finalisation and Submission for Migration

After expert validation, the microservice recommendations are integrated and prepared for handoff. The process combines the Architectural Reasoning Context (ARC), validated microservice groupings (in JSON), and expert review notes into a single summary. The second LLM agent then uses this JSON output and generates human-readable report. This report includes microservice names, justification narratives, and applied rule ID.

The final output of this approach serves as the microservice identification proposal, in which the recommendations are submitted to architects for further execution. It ensures that the transition is guided by structured reasoning, aligned with expert decisions, and ensures compliance with policy, combining AI-driven insights with practical implementation readiness for system decomposition.

EXPERIMENT AND RESULTS

In this section, we present the experimental evaluation of the proposed AI-assisted approach to identify the candidate microservices using large language models and Architectural Reasoning Context (ARC). The evaluation focuses on validating the approach in terms of its accuracy and architectural rule compliance in two software domains: Infra Event Management and an E-commerce platform.

Two different types of architectural patterns were selected, transaction-oriented and event oriented. This allows comparison to evaluate how well the proposed method is applied across different service interaction patterns.

The evaluation in this study validates the proposed method's architectural reasoning, rather than a large-scale predictive benchmark. Microservice boundary identification is an inherently design-oriented and context-specific process. Therefore, the domain-specific controlled simulations provide a suitable platform for determining structural correctness, rule compliance, and expert judgement.

Although full system-level implementation is beyond the current scope, the pipeline was prototyped and validated through a structured simulation of its core components. ARC artefacts were manually constructed from representative SOA-style systems, and LLM reasoning was performed using structured prompts via the OpenAI API (GPT-5). The selected model was chosen for its accuracy in structured reasoning tasks, shows conformity to structured output formats like JSON, and is capable of processing large contextual inputs. These qualities are critical for effectively handling the complexities of ARC representation. Expert reviewers validated the microservice groupings generated based on rule compliance and domain alignment. This simulation-based evaluation demonstrates the feasibility, architectural reasoning capabilities, and practical utility of the proposed method.

To ensure reproducibility of experimental results, the same set of prompt templates, ARC structures, and configuration parameters (i.e., temperature and token limit) were utilised in each experimental run.

The evaluation therefore focuses on validating the structural integrity of the proposed service boundaries and ensuring adherence to architectural policies rather than attempting to fully automate expert decision-making.

Experimental Configuration

Two domain-specific synthetic applications that simulate real-world service-oriented architectures were used to evaluate the proposed approach. The domains include:

1. An E-commerce application capturing patterns of transactional, service-oriented systems.
2. An Infra Event Management application capturing the patterns of event ingestion, correlation, incident response, and on-call notifications in service-oriented infrastructure.

The domains were selected for evaluation represent transaction-heavy and event-driven architectural styles. Each experiment was evaluated over three iterations with the same configuration to compare the robustness and consistency of the outcomes.

Each system contained approximately 10–15 service interfaces (e.g., WSDL, OpenAPI) , each exposing multiple public methods and 80–120 class definitions, from which candidate functional units were selected using the heuristics described in Section 3.2.1.1 The structural relationships, dependencies between interfaces, and modular responsibilities were obtained statically from the source code to simulate realistic SOA environments. This metadata served as the basis for constructing ARC.

For each domain, a representative set of public methods (functional units) was selected, and a corresponding set of service boundary identification rules was instantiated to capture the major functional areas and architectural policies of the underlying SOA systems, as summarised in Table 3.

Table 3. Experimental configuration of functional units and service boundary identification rules across the two evaluation domains

Domain	Number of Functional Units	Total Number of Rules	Conflict Rules Pairs
E-commerce	24	20	2
Infra Event Management	45	28	3

Measures of Evaluation

To evaluate the performance of proposed approach, the following evaluation measures were applied.

Net Accuracy (%)

This metric measures the accuracy of the LLM suggested microservices. Each microservice receives +1 if it is correctly suggested and -1 if the recommendation is wrong or missing (i.e., a microservice that is wrongly suggested or not recommended by the proposed method). The total score is then normalised by the number of expert-defined microservices and reported as a percentage.

$$\textbf{Net Accuracy}\ (\%) = \frac{\mathbf{N_{cm}}}{\mathbf{N_m}} * \mathbf{100} \qquad (\mathbf{1})$$

Where:

- $\mathbf{N_{cm}}$ is the net microservice credit score of the proposed method, computed as (+1) for each correctly suggested microservice and (−1) for each wrong or missing microservice.
- $\mathbf{N_m}$ is the total expert-defined microservices.

Rule Compliance (%)

This metric indicates the percentage of service boundary identification rules that are correctly applied by the LLM-guided method while generating candidate microservices. A rule is counted as correctly applied when it is satisfied or explicitly relaxed due to a detected conflict under the proposed rule-precedence strategy. A rule is counted as not complied when it is violated without being relaxed by the precedence strategy. Rule Compliance Rate is computed as the ratio of correctly applied (including relaxed) rules to the total number of rules and is reported as a percentage.

$$\textbf{Rule Compliance}\ (\%) = \frac{\mathbf{N_{cr}}}{\mathbf{N_r}} * \mathbf{100} \qquad (\mathbf{2})$$

Where:

- $\mathbf{N_{cr}}$ is the number of correctly applied rules including, those relaxed due to conflict by proposed method.
- $\mathbf{N_r}$ is the total number of rules.

Together, these measures quantify both structural accuracy (Net Accuracy) and policy alignment (Rule Compliance), providing a comprehensive and balanced evaluation of the proposed method's effectiveness.

Findings and Analysis

The proposed approach evaluation was conducted on service-oriented architecture (SOA) applications from the E-commerce and Infra Event Management domain. This evaluation examined how effectively the proposed method identifies microservice boundaries in line with expert expectations, while also ensuring the legitimacy of defined architectural rules. A summary of the key findings is presented below:

Net Accuracy (%)

The proposed approach attained average Net Accuracy of 66.66 % for the E-Commerce domain and 70.37 % for the Infra Event Management domain. These results demonstrate that the generated microservice show substantial alignment with expert-defined groupings, indicating that it identifies correct service boundaries in the majority of cases.

Rule Compliance (%)

E-Commerce and Infra Event Management cases achieved average Rule Compliance Rates of 76.66% and 73.80%, respectively. This indicates the LLM consistently used service boundary identification rules.

Table 4 shows the evaluation findings of the proposed method tested on two representative applications: E-commerce and Infra Event Management

Table 4. Evaluation Results of the AI-Assisted approach.

Domain	Iteration	Ncm	Nm	Net Accuracy (%)	Ncr	Nr	Rule Compliance Rate (%)
E-commerce	1	6	10	60.00	14	20	70.00
E-commerce	2	7	10	70.00	16	20	80.00
E-commerce	3	7	10	70.00	16	20	80.00
Infra Event Management	1	12	18	66.67	20	28	71.42
Infra Event Management	2	13	18	72.22	21	28	75.00
Infra Event Management	3	13	18	72.22	21	28	75.00

Table 5 shows average of findings of the proposed approach that are tested on two domains, E-commerce and Infra Event Management.

Table 5 Average Evaluation Results of the Approach.

Domain	Net Accuracy (%)	Rule Compliance Rate (%)
E-Commerce	66.66	76.66
Infra Event Management	70.37	73.80

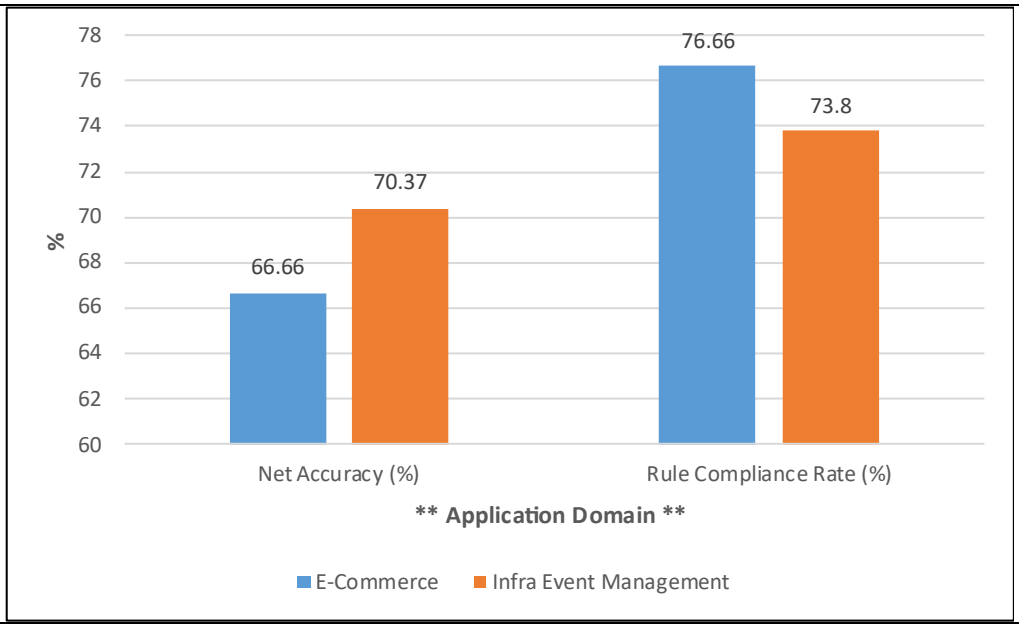

Fig 9. Evaluation Results Across Domains.

Figure 9 compares the proposed method's average performance across two representative domains: e-commerce and Infra Event Management. The findings substantiate that the proposed approach yields

technically valid, domain-aware, and explainable microservice boundaries. Its relevance to real-world migration scenarios is demonstrated by consistent agreement with experts' expectations throughout the evaluations.

This gap arises because rule compliance is assessed locally at the level of individual constraints applied to specific functional_units, whereas Net Accuracy is evaluated globally by comparing entire microservice groupings with a single expert reference service boundary configuration. In many cases, microservices classified as “inaccurate” still satisfy all applicable rules but diverge from the expert solution in how borderline units are merged or split across neighbouring services. Consequently, even with only moderate Net Accuracy values, the proposed method remains appropriate as an AI-assisted decision-support mechanism that reduces manual effort for architects while keeping final boundary decisions under expert control.

Analysis and Validity Risk

In interpreting the findings, it is necessary to recognise a number of practical limitations, despite positive findings in terms of the automation of candidate microservice identification for SOA-to-MSA migration utilising structured context and LLMs.

Effects of Real vs. Synthetic Data

Synthetic systems were good for control and repeatability; however, they do not represent the variability and unpredictability of a real-world SOA environment such as undocumented dependencies, and architectural inconsistencies. Future validation of the proposed method on real open-source or production systems is suggested.

Reliance on Input Artefact Completeness

The approach is dependent upon accurate and complete code architectures, declarations of services, and well-defined service boundary rules. This may present challenges when relying on the use of legacy systems with input artefacts that are incomplete, or unavailability of practitioner domain experts to properly determine the necessary input, all of which could potentially compromise ARC quality, and rule enforcement.

Variability in LLM Output and Prompt Sensitivity

The LLMs tend to produce varying results following changes to prompt configuration, such as phrasing, "temperature," or model version.

Throughout the experiment, multiple patterns of prompt-sensitivity were observed. Minor changes to the prompt phrasing such as re-ordering the instructions, changing the connectors and moving from high level of explanation to minimal level (in examples), primarily affected non-functional aspects, including (but not limited to) the format of the microservice names and language associated with the justification, while having little effect on the assignment of functional_units. However, more substantial modifications to the prompts have produced varying treatment of borderline functional units. In many cases, functional units that were previously viewed as separate were merged into one subdomain or split into several. Experiments using the same template, a low temperature setting and constant LLM model configuration, produced very similar results for most of the groupings of all the identified functional_units. Variability, if any exists, is limited to a few of the more ambiguous units. As a result, this study utilising the prompt configuration and model settings will be able to reproduce the overall patterns of service boundary identification, however, the prompt configuration will also impact potential ambiguities. Therefore, all experimental findings are based on versioned prompts to allow for replication and direct comparison between studies.

CONCLUSION & FUTURE WORK

This research paper presented a lightweight expert-guided, AI-assisted approach for identifying microservices from legacy SOA systems. The approach leverages large language models (LLMs) to assist in architectural reasoning and produce easy-to-interpret groupings of microservices, guided by structured prompts, Architectural Reasoning Context (ARC) in JSON format and well-defined service boundary rules.

By integrating multiple static input artefacts such as source code, service interface definitions, and expert-defined rules, the approach supports architecture-aware decision-making with human-in-the-loop validation, enabling explainable and policy-compliant microservice identification.

domain-specific systems, Infra Event Management and e-commerce. These systems demonstrated consistent structural accuracy, consistent adherence to rules, and alignment with expert expectations. The structured design of valid prompts and incorporating domain-specific constraints improved reliability and interpretability of LLM outputs while preserving the overall architectural integrity, reducing manual effort.

Overall, the proposed method bridges domain-aware architectural reasoning with AI capabilities, offering a practical and extensible foundation for supporting intelligent legacy system modernization. It promotes

explainable, policy-compliant microservice identification, particularly valuable in enterprise and regulated environments.

Future enhancements to the approach may involve expanding the scope and improving the intelligence of service boundary identification by incorporating additional artefacts. These artefacts include communication logs, database schemas, and configuration files. This would enrich the Architectural Reasoning Context (ARC) by enhancing boundary detection, data flow models, and implementation of service interaction. Further directions will include support of multilingual code assets, integration with fine-tuned LLMs trained on software architecture corpora which enables iterative learning loops to collect and utilise real-time feedback from evolving systems. A significant enhancement would involve shifting from a GenAI-centric reasoning pipeline to an agentic architecture. An agentic architecture would enable creation of autonomous orchestration among specialised agents, such as ARC builders, rule validators, and report generators. This facilitates goal-driven, context-aware, and explainable microservice design with continuous and active long-term memory and dynamic control. Additionally, a full system-level implementation, integrated into an end-to-end toolchain, is planned to support production deployment and practical adoption.

The current study relies on prompt-engineered use of a general-purpose LLM. As future work, the model could be fine-tuned using domain knowledge and real SOA-to-MSA migration datasets, enabling more accurate and consistent microservice boundary recommendations beyond prompt-based reasoning.

REFERENCES

De Alwis, A.A.C., Barros, A., Fidge, C., & Polyvyanyy, A. (2019). Business object centric microservices patterns. In Springer eBooks (pp. 476–495). DOI:10.1007/978-3-030-33246-4_30

Henry, A., & Ridene, Y. (2019). Migrating to microservices. In Microservices (pp. 45–72). DOI:10.1007/978-3-030-31646-4_3

Garriga, M. (2018). Towards a taxonomy of microservices architectures. In Software Engineering and Formal Methods (pp. 203–218). DOI:10.1007/978-3-319-74781-1_15

Chen, R., Li, S., & Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. In 2017 24th Asia-Pacific Software Engineering Conference (APSEC). DOI:10.1109/apsec.2017.53

Li, S., et al. (2019). A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software, 157, 110380. DOI:10.1016/j.jss.2019.07.008

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2018). Challenges when moving from monolith to microservice architecture. In Current Trends in Web Engineering (pp. 32–47). DOI:10.1007/978-3-319-74433-9_3

Nunes, L., Santos, N., & Rito Silva, A. (2019). From a monolith to a microservices architecture: An approach based on transactional contexts. In Lecture Notes in Computer Science (pp. 37–52). DOI:10.1007/978-3-030-29983-5_3

Bucchiarone, A., Soysal, K., & Guidi, C. (2020). A model-driven approach towards automatic migration to microservices. In Lecture Notes in Computer Science (pp. 15–36). DOI:10.1007/978-3-030-39306-9_2

Pigazzini, I., Arcelli Fontana, F., & Maggioni, A. (2019). Tool support for the migration to microservice architecture: An industrial case study. In Lecture Notes in Computer Science (pp. 247–263). DOI:10.1007/978-3-030-29983-5_17

Khadka, R., Saeidi, A., Jansen, S., Hage, J., & Haas, G. P. (2013). Migrating a large-scale legacy application to SOA: Challenges and lessons learned. In 2013 20th Working Conference on Reverse Engineering (WCRE). DOI:10.1109/wcre.2013.6671318

Abdullah, M., Iqbal, W., & Erradi, A. (2019). Unsupervised learning approach for web application auto-decomposition into microservices. Journal of Systems and Software, 151, 243–257. DOI:10.1016/j.jss.2019.02.031

Ma, S.-P., Fan, C.-Y., Chuang, Y., Liu, I.-H., & Lan, C.-W. (2019). Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems, 100, 724–735. DOI:10.1016/j.future.2019.05.048

Escobar, D., et al. (2016). Towards the understanding and evolution of monolithic applications as microservices. In 2016 XLII Latin American Computing Conference (CLEI). DOI:10.1109/clei.2016.7833410

Abgaz, Y. M., et al. (2023). Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering, 1–32. DOI:10.1109/tse.2023.3287297

Oumoussa, I., & Saidi, R. (2024). Evolution of microservices identification in monolith decomposition: A systematic review. IEEE Access, 1–1. DOI:10.1109/access.2024.3365079

Narváez, D., Battaglia, N., Fernández, A., & Rossi, G. (2025). Designing microservices using AI: A systematic literature review. Software, 4(1), 6. DOI:10.3390/software4010006

Alsayed, A. S., Dam, H. K., & Nguyen, C. (2024). MicroRec: Leveraging large language models for microservice recommendation (pp. 419–430). DOI:10.1145/3643991.3644916

Sellami, K., & Saied, M. A. (2025). MonoEmbed: Enhancing LLM representations for monolith to microservices decomposition through contrastive learning. Empirical Software Engineering, 31(1). DOI:10.1007/s10664-025-10732-z

Wang, Y., Bornais, S., & Rubin, J. (2024). Microservice decomposition techniques: An independent tool comparison. In 2024 IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1295–1307). DOI:10.1145/3691620.3695504

Trabelsi, I., Moha, N., & Guéhéneuc, Y.-G. (2025). Exploring the systematic use of LLMs for microservices generation. In Lecture Notes in Computer Science (pp. 121–128). DOI:10.1007/978-981-96-7238-7_10

De Alwis, A.A.C., Barros, A., Fidge, C., & Polyvyanyy, A. (2019). Availability and scalability optimized microservice discovery from enterprise systems. In Lecture Notes in Computer Science (pp. 496–514). DOI: 10.1007/978-3-030-33246-4_31

Cite this article as: Sandeep Sharma(2026). The Role of Big Data in Advancing Artificial Intelligence: Methods and Case Studies. *International Journal of Artificial Intelligence and Machine Learning*, 6(1), 150-167. doi: 10.51483/IJAIML.6.1.2026.150-167.