www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(4), 2026, 233-251. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2026.v17i4.1434

Method for migrating monolithic software systems and their global database to isolated microservices

Omar Hernández López[1], René Santaolaya Salgado[1], Blanca D. Valanzuela Robles[1], Noé A. Castro Sánchez[1], Juan Carlos Rojas Perez[1], Sergio F. Ruiz Paz[2]

[1]TecNM Centro Nacional de Investigación y Desarrollo Tecnológico, México
[2]Universidad del Papaloapan, México
Email address(es): {d16ce079, rene.ss, blanca.vr, juan.rp, noe.cs}@cenidet.tecnm.mx, sruiz@unpa.edu.mx

Abstract. Migrating monolithic applications to microservice architectures offers benefits such as scalability, resilience, and continuous deployment. However, decomposing shared monolithic databases remains a major challenge due to strong structural and functional dependencies among application components and shared relational schemas. This paper proposes a semi-automatic decomposition method that identifies candidate microservices and service-specific databases through static analysis of use-case interaction sequences, source code dependencies, and database access relationships aligned with business capabilities. The proposed approach evaluates structural coupling using the Shared Table Coupling Index (IATC) indicator before and after migration. The method was validated through a case study and an empirical evaluation involving 17 monolithic systems from public repositories. The results show a statistically significant reduction in shared-data coupling, with an average reduction of approximately 91%, indicating the effectiveness of the proposed approach for reducing structural dependencies during monolith-to-microservices migration.

Keywords: monolithic systems, microservices migration, database decomposition, database-per-service, static analysis.

Article information
Received: May 8, 2026
Accepted: Jul 11, 2026

1 Introduction

The transition from legacy monolithic systems to microservices architectures is one of the main challenges for modernizing enterprise applications in operation. In particular, the decomposition of the monolithic database using the "database per service" approach has been extensively studied in the literature (Laigner et al., 2021).

Decomposing the monolithic database presents a critical problem, as monolithic systems often exhibit a high level of indirect coupling due to shared access to a single data store (André et al., 2025). Solving this problem involves correctly identifying the boundaries of each microservice and, at the same time, defining, for each boundary, the data model that each microservice should manage independently, thus generating a separate database for each one.

Several approaches for migrating monolithic systems to microservices identify service candidates through techniques such as static analysis, dynamic analysis, software repository mining, clustering, and dependency analysis. However, these approaches mainly focus on decomposing application logic and defining service boundaries. The decomposition of the underlying database is often overlooked, even though the shared database remains one of the main sources of coupling in monolithic systems.

To address this limitation, this research proposes a detailed method for decomposing monolithic legacy systems into their corresponding microservices, including the segregation of the monolithic database present in these systems. The method focuses on the lexical-syntactic analysis of the original source code using the ANTLR (Another Tool for Language Recognition) metalanguage and the Visitor pattern. These mechanisms are used to traverse the static structure of the application and identify classes, methods, and their relationships. The resulting relationships are organized into interaction sequences associated with the realization of specific business functionalities (Parr, 2013).

In this work, an interaction sequence is defined as an ordered static traversal of class-method invocations, service interactions, and data access relationships associated with the realization of a specific use case or business functionality. The implicit data model refers to the set of entities, attributes, and relationships that can be inferred from the source code and persistence structures, even when no explicit conceptual data model is available. By traversing the interaction sequences, the proposed method reconstructs this implicit data model and identifies microservice candidates together with their corresponding databases. The information resulting from the analysis is stored in a data structure designed to represent and regenerate each candidate microservice, its associated database, database scripts, and deployment artifacts.

Existing migration approaches frequently rely on runtime information, such as execution traces, logs, or monitoring data. However, such information is often unavailable in legacy systems. To address this limitation, the proposed method identifies microservice candidates and associated databases using only static source-code analysis. This allows the method to be applied even in environments where controlled execution and behavioral monitoring are not feasible (Krause et al., 2020; Andrade et al., 2023; Jin et al., 2021).

Another limitation of existing approaches is their primary focus on service boundary identification while providing limited support for database decomposition. To overcome this issue, the proposed method jointly identifies microservice candidates and their corresponding databases by reconstructing the implicit data model from interaction sequences. This contributes to reducing indirect coupling caused by shared persistence and facilitates the adoption of the database-per-service principle.

In addition, the parsing infrastructure is designed to be adaptable to multiple programming languages. Since the method is based on ANTLR grammars, it can be extended to different technological environments with limited modifications to the analysis process.

To evaluate the effectiveness of the proposed decomposition process, this work introduces the Index of Shared Table Coupling (IATC). The IATC is a metric designed to quantify the degree of coupling caused by shared database tables among microservices. Lower IATC values indicate fewer shared persistence dependencies and a higher degree of database autonomy after decomposition.

The method also addresses two practical challenges commonly reported in monolith-to-microservice migration studies:

- Scalability of the analysis, through a static process that can be applied to large software systems with limited human intervention.
- Empirical validation, through the implementation of the method in a software tool and its evaluation using open-source systems to assess its effectiveness and generalizability.

Overall, the method contributes to the automation of monolithic system migrations by offering a systematic and reproducible process for identifying microservices and their associated databases within legacy code. This facilitates the adoption of flexible, scalable, autonomous architectures aligned with organizational goals, while simultaneously laying the groundwork for future extensions to modern technological environments.

The remainder of this paper is organized as follows. Section 2 presents the related work, reviewing systematic literature reviews and existing methods for monolith-to-microservice decomposition. Section 3 describes the proposed methodology, detailing each step of the decomposition process, including lexical-syntactic analysis, interaction sequence extraction, and database partitioning. Section 4 presents the software tool developed to support the implementation of the method. Section 5 applies the proposed method to a real-world case study, illustrating the end-to-end decomposition process. Section 6 describes the experimental evaluation. Section 7 presents the results analysis using the Index of Shared Table Coupling (IATC), a metric proposed to quantify the coupling generated by shared database tables among the identified microservices. The IATC evaluates the reduction of database-related coupling by comparing the shared access to database tables before and after the decomposition process. Finally, Section 8 discusses future work.

2 Related Work

The transition from monolithic systems to microservices architectures has been extensively studied, particularly from the perspectives of design, architectural patterns, and refactoring strategies (Newman, 2015), (Lewis & Fowler, 2014). However, the decomposition of the monolithic database remains a critical challenge and one of the least addressed areas in the scientific

literature. While the principles of "database per service" are well established, systematic, reproducible, and semantically sound software methods still have significant gaps.

Early approaches proposed conceptual or heuristic methods for database decomposition, largely based on business domains, bounded contexts, or modular separation strategies (Evans, 2004), (Richardson, 2018). However, these approaches rely heavily on expert knowledge and do not offer a detailed procedure for deriving databases from monolithic system artifacts such as classes, methods, services, call sequences, or data access patterns.

Abgaz et al. (2023) rigorously examined 35 research articles using a systematic literature review protocol and snowballing, proposing the Monolith to Microservices Decomposition Framework (M2MDF). This framework identifies the main phases and key elements of decomposition, together with a detailed analysis of existing approaches, tools, and methods. Their findings suggest that monolith-to-microservices decomposition is still in its early stages and that there is a notable lack of methods that combine static, dynamic, and evolutionary data. This gap directly motivates the present work, which operates exclusively on the static analysis of source code.

Hassan et al. (2024) conducted a systematic literature review to explore the various methodologies, techniques, and algorithms used in migrating monolithic systems to modern microservices architectures. Their study highlights the role of artificial intelligence in improving the efficiency and effectiveness of the migration process, while identifying significant patterns, challenges, and optimal solutions. This work reinforces the relevance of semi-automated approaches to address the complexity of migration processes.

Castillo-Estrada et al. (2025) present a systematic mapping of the literature on relational database decomposition methodologies for the migration from monolithic systems to microservices. They conclude that this field is still in an early stage and that no standardized methodologies currently exist to enable efficient migration at the data-layer level.

Mazlami et al. (2017) propose a method for extracting microservices from monoliths by analyzing Git repository history and identifying highly cohesive modules through three coupling strategies: logical coupling, semantic coupling, and contributor coupling. Logical coupling analyzes revision history to determine which classes change together, semantic coupling examines class vocabulary, and contributor coupling focuses on code ownership derived from repository history. However, their approach does not consider database structures or data access patterns.

In industrial tools, IBM Mono2Micro proposes a decomposition approach called “space-time,” which combines two dimensions of analysis: (1) *space*, corresponding to the code structure obtained through static analysis (classes, dependencies, hierarchies, and packages), and (2) *time*, derived from execution traces that capture how classes interact across different use cases. By integrating both sources, Mono2Micro identifies components that are not only structurally related but also interact at runtime, potentially increasing the functional cohesion of the proposed microservices. Furthermore, the tool automatically generates the necessary communication contracts between the resulting partitions (Kalia et al., 2021).

However, this reliance on dynamic information contrasts with purely static approaches, such as the one proposed in this work, which operate solely on the source code and data model without requiring trace collection. Static methods offer lower operating costs, greater reproducibility, and applicability even when an executable environment is unavailable.

Recent research has progressed toward automatic and semi-automatic approaches based on static analysis, clustering, dependency extraction, and machine learning. In graph modeling studies, several works represent classes as nodes and dependencies (calls, data access by inheritance) as edges to apply clustering and combinatorial optimization, aiming to maximize cohesion and minimize coupling between services (Filippone et al., 2023).

Recent approaches have evolved toward deep learning architectures based on Graph Neural Networks (GNNs). In particular, hybrid models that use a combination of a Variational Autoencoder and a Graph Neural Network, such as VAE-GNNs, combined with C-means clustering algorithms, utilize dependency graphs obtained through static analysis and coexistence matrices of input points to capture both the structure and functional use of the system. These models generate representations that suggest partitions with greater cohesion (Maharjan et al., 2025). Our approach, which infers database candidates per service from interaction sequences of classes participating in the realization of specific business functionalities, fits naturally into this graph representation, but is distinguished by prioritizing the data model implicit in the sequences, aligning with a database per service.

Specifically regarding database decomposition, the literature acknowledges the challenge of transferring data ownership and breaking shared foreign keys. There are progressive migration patterns and guides (e.g., database-per-service and pattern chains for moving data ownership) that document partitioning and synchronization steps during the transition (Richardson, 2018), (Volynsky et al., 2022). Furthermore, more comprehensive bibliographies such as (Ayas et al., 2023) and SLR empirical studies consolidate phases, metrics, and open gaps (e.g., scarcity of benchmarks and standardized datasets), reinforcing the need for automated methods based on static artifacts, such as the one presented in this research.

Hao et al. (2023) proposed a relational database decomposition method oriented to microservice refactoring. Their approach focuses on partitioning relational schemas into service-specific databases in order to reduce dependencies between services during migration. However, the work mainly emphasizes database schema decomposition and provides limited support for integrating interaction sequences, business-oriented use-case analysis, and comprehensive source-code dependency analysis, which are central aspects of the present work.

Finally, (Freitas et al., 2023) propose a methodology for automatically refactoring monolithic web applications based on object-relational mapping (ORM) into microservices architectures, taking the source code and an initial microservice proposal as input. The methodology creates an API for each method call of classes belonging to other services, and the database entities are refactored to be included in the corresponding microservice. This approach is similar to the present work in terms of deriving database schemas per service from the relationships between classes; however, it assumes that all database information is contained in ORM annotations in the code, and requires a previously defined microservice proposal as input.

Unlike previous approaches that focus independently on schema partitioning, dynamic trace analysis, clustering techniques, or ORM-based refactoring, the proposed method jointly reconstructs static interaction sequences, infers implicit data dependencies, and derives service-specific databases directly from source code artifacts aligned with business functionality.

Table 1. Comparison of related works and contribution of the proposed approach

Author	Approach	Inputs	Limitations
Maharjan et al. (2025)	Deep learning: VAE-GNN + C-means clustering	Dependency graphs and entry point coexistence matrices	Does not consider the data model; requires model training
Hao et al. (2023)	Relational database decomposition	Relational schemas and service boundaries	Focuses mainly on schema partitioning; does not incorporate interaction sequences or static code analysis.
Freitas et al. (2023)	Static analysis + code refactoring	Packages and ORM entities	Requires initial manual decomposition.
Kalia et al. (2021)	Static + Dynamic analysis + Machine Learning	Class cohesion	Does not generate a database per service; requires training.
Mazlami et al. (2017)	Static analysis + Machine Learning	Cohesive components	Does not consider the data model; requires supervised clustering.
This Work	Static analysis: AST + interaction sequences + data access extraction	Legacy monolithic system	The implementation tool requires adaptation to the target programming language.

The development of an integrated methodology is proposed in this work. It combines static code analysis, interaction sequences between methods, data model analysis, and mechanisms for ensuring information consistency through patterns such as the observer and event-driven approaches. This methodology represents a significant contribution toward overcoming the identified limitations and provides a more general, robust, and automated solution for data model decomposition in microservices migration.

3 Research Methodology

The proposed method is organized into three main phases. In the first, the client accesses the monolithic system by providing the physical path to its source code. The method then traverses the input directory recursively, validating each file according to its extension and reading its internal structure packages, classes, methods, and invocation relationships without modifying the original code. Language independence is achieved through ANTLR grammars adapted to the target language (e.g., Java.g4, CSharp.g4, Python3.g4).

In the second phase, a lexical-syntactic analysis is performed using ANTLR to generate a parse tree for each file, from which an Abstract Syntax Tree (AST) is constructed. The Visitor pattern traverses the AST and extracts structural elements such as class annotations (@Entity, @RestController, @Repository), fields, methods, method bodies, and inter-class invocations, storing them in a syntactic data structure for subsequent processing. Simultaneously, interaction sequences are identified by tracing public method calls in public classes, which correspond, to actor requests forming chains of invocations that represent individual use-cases. During this traversal, the persistence entities accessed by each method are identified, thereby associating a data model with each use case candidate. Cross-references between files are also resolved at this stage to ensure completeness of each candidate's data model.

In the third phase, shared entities, those accessed by more than one microservice candidate are detected, and each service is assigned its own copy of the shared data within its bounded context, applying the Database-per-Service pattern. Finally, a complete set of implementation artifacts is produced for each microservice, including domain classes, service and repository layers, REST controllers, observer and publisher classes, SQL scripts for database creation, and deployment configuration files.

Figure 1 presents the BPMN process diagram of the proposed migration method, illustrating the three main processes involved in decomposing a monolithic system and its database into isolated microservices. Each process and its constituent steps are described in detail in the following subsections

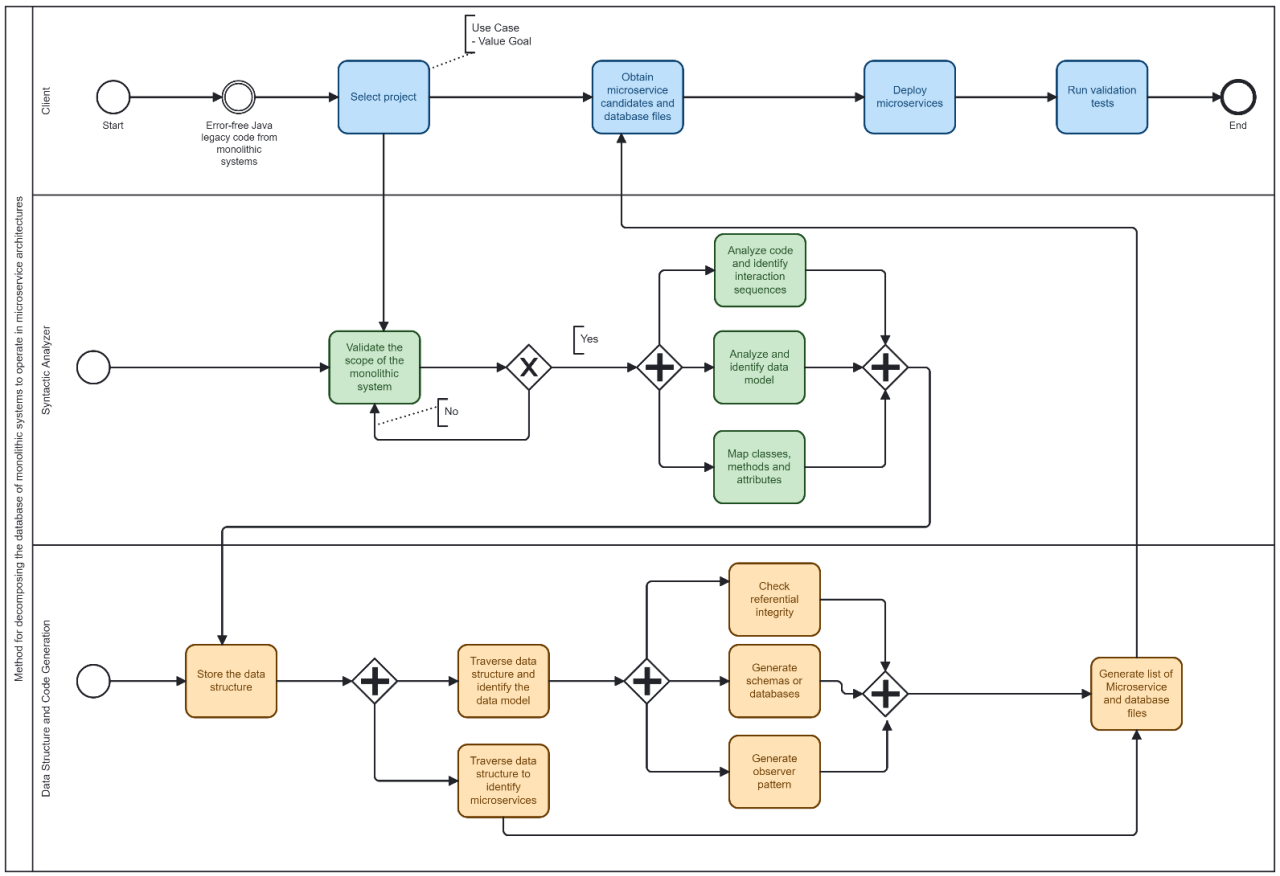

Fig. 1. BPMN diagram of the monolithic system migration method and database decomposition per microservice.

A use-case, in the context of object interaction, can be realized as a sequence of messages between collaborating objects (Booch et al., 1999), (Kimmel, 2008). For each message from the client object, there is a corresponding method response from the server object. When one object passes a message to another, the receiving object forwards it successively, forming a message flow that constitutes a sequence. In object-oriented systems, business functionalities are typically realized through coordinated sequences of method invocations among collaborating classes. Based on this principle, the proposed method statically reconstructs interaction sequences by tracing invocation relationships initiated from externally exposed service entry points. These sequences allow the identification of the persistence entities accessed during the realization of a business functionality, enabling the derivation of candidate microservices and their associated service-specific databases.

Use cases are identified by analyzing the public methods of public classes, which represent actor requests, and in modern technologies, through annotations; for example, in java-controllers (@controller). Therefore, public method invocations can be considered the starting point for identifying interactive message sequences that implement use cases. A public method of a public class initiates an interaction sequence of method calls that stops when the last called method returns process control.

Figure 2 presents an example of interactive message sequences representing two use cases in a system and their access to the data layer. In use case UC-1, the method $m_1 \rightarrow m_{1a} \rightarrow m_{1b}$ ends with a call to m_{1d}, which represents the Order and Inventory tables. The sequence begins with the method $m_1 \rightarrow m_{1e} \rightarrow m_{1c}$ and ends with a call to the method m_{1d}, which registers the order. Method m_{1a} accesses the Customer table, method m_{1b} accesses the Product table, method m_{1e} accesses the Inventory table, and method m_{1c} accesses the Order table. When persistence is performed with method m_{1d}, the Order and Inventory tables are updated, respectively. Use case UC-2 cancels the order and affects the Inventory table, which is shared by both use cases.

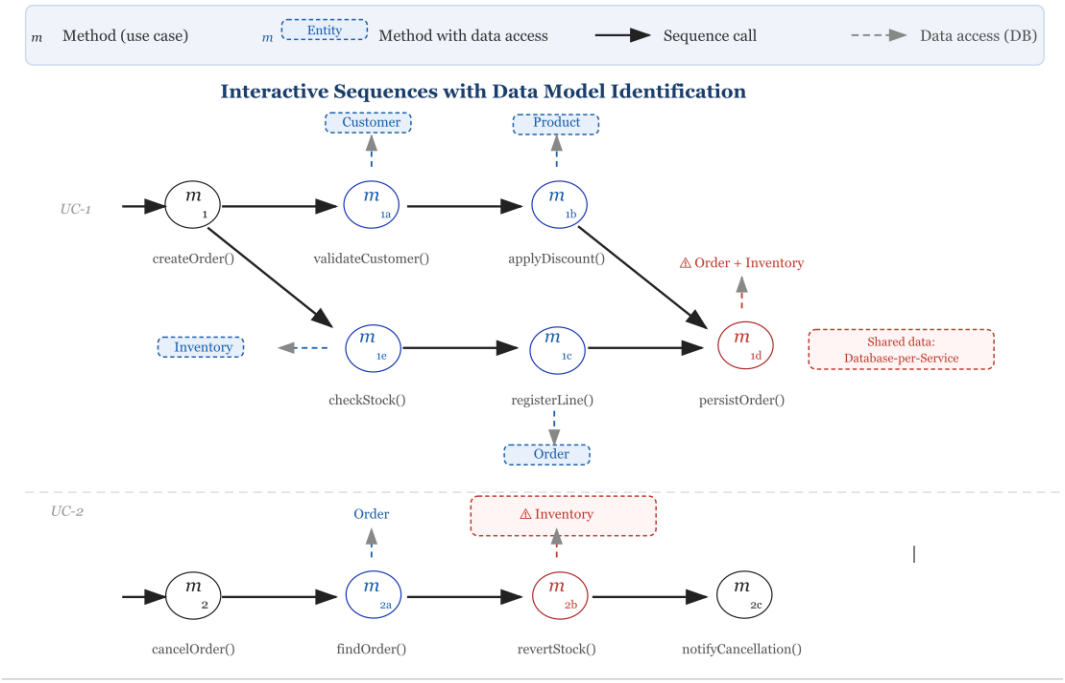

Fig. 2. Interaction sequences of two use cases with data model identification per method.

Having established the conceptual basis of use case identification through interaction sequences, the following subsections describe each step of the proposed migration method in detail.

3.1 Access and review of the monolithic system

The monolithic system must be written in an object-oriented programming language, such as Java, which must be compiled and free of syntax errors. First, access the physical path containing the system's source code (e.g., c:/project/...). Code preprocessing involves traversing the input path to detect and validate each file according to its extension. Each file in the monolith is read recursively, identifying its internal structure packages, classes, methods, and invocation relationships without modifying the original code.

To ensure language independence, this step relies on ANTLR grammars adapted to the target language (e.g., Java.g4, Python3.g4, or CSharp.g4), allowing the analysis method to be applied in any programming language.

3.2 Lexical-syntactic analysis and construction of the AST

For parsing, the method suggests using ANTLR with a stable grammar (e.g., the java8 grammar from grammars-v4). Each file is transformed into a character stream and processed by the ANTLR-generated lexer and parser to obtain a parse tree. From this, an Abstract Syntax Tree (AST) is constructed: The visitor pattern also extracts and preserves textual fragments (e.g., complete method bodies) and semantic metadata, which are stored in the syntactic data structure for subsequent processing.

The implemented visitor pattern extends the ParserBaseVisitor class also generated during compilation and, for each compilation unit, extracts the elements that identify a class and its sequence, for example, in Java language:

- Package declaration and imports.
- Class annotations (@Entity, @Table, @RestController, @Service, @Repository).
- Fields (type, name, annotations).
- Methods (signature, annotations, body text).
- Detected invocations (method Call) with minimal context (text invoker and method name).
- Local declarations that allow mapping variable names.

The extraction of the method body is performed using the token indices provided by ANTLR to preserve the original semantics exactly and allow for the regeneration of text stored in a data structure. The AST process allows the construction of interaction sequences that describe how the system's classes collaborate to satisfy a specific use case. Each sequence is modeled as a chain of invocations between components, starting with a public class and identified by annotations (e.g., Controller, Service, Repository, Entity), thus representing the smallest functional unit that fully and adequately fulfills a value goal.

Analyzing these sequences is essential for identifying microservice candidates, since each self-contained functional flow that starts in a public class and ends in the manipulation of data entities can be considered the basis of an independent service. This process can be represented using automatically generated graphs, where the interaction between components and the sharing of data entities are visualized.

3.3 Identification and Decomposition of the Data Model

Once the interactive sequences have been identified, the relationship between the classes and the persistence entities with the data model is analyzed. From this analysis, the data model dependencies are extracted, identifying which entities are manipulated by each sequence. These entities form the microservices data model, guaranteeing the principle of generating a database per service.

In cases where multiple microservices require access to the same table or entity, each service must maintain its own copy of the data within its bounded context. To support this, an eventual consistency strategy based on the Observer/Events or messaging pattern should be implemented. In this approach, microservices subscribe to events published by the service that owns the data. This strategy avoids direct database coupling and allows state changes to propagate across microservices through events.

The architectural principle underlying the proposed database decomposition approach is the Database-per-Service pattern, which establishes that each microservice must exclusively own and manage its own data store, without sharing database schemas or instances with other microservices. This principle is essential to achieve deployment independence, operational autonomy, and fault resilience in microservice architectures.

However, strictly applying this principle creates a challenge when multiple business operations in a monolithic system access the same domain entity. In such cases, assigning the entity to a single microservice would force other services to depend on external data ownership, reintroducing the structural coupling that the migration process aims to eliminate. Therefore, the proposed method addresses this issue through controlled entity duplication and event-based synchronization. This strategy is implemented through the following database decomposition rules.

To guide the database decomposition process, a set of transformation rules was defined to identify shared entities, assign ownership, and manage data replication among microservices. These rules are organized into three groups.

The first group focuses on the detection of shared entities. An entity is considered shared when it is accessed by more than one microservice candidate. Additionally, transitive relationships between entities are analyzed to identify indirectly shared entities through the data model.

The second group defines entity ownership among microservices. Ownership is assigned based on business operations that create, modify, or delete data associated with an entity. Microservices that only perform read operations are considered consumers and receive a replicated version of the entity. When no direct owner exists, ownership is assigned according to the entity's relevance within the business context and its access patterns, which commonly corresponds to reference or catalog data.

The third group establishes replication and consistency rules. Shared entities are replicated in the databases of consuming microservices to preserve autonomy and avoid cross-database dependencies. Each microservice maintains an independent schema, and relationships between services are represented through identifiers instead of foreign keys. When the owner microservice modifies an entity, it emits a change notification, allowing consumer services to update their local replicas and maintain eventual consistency.

These rules enable the transformation of a shared monolithic database into independent service-specific databases while reducing coupling caused by shared persistence.

3.4 Automatic Generation of Microservices Code

The system automatically generates the identified microservices, creating a complete structure for each one that includes:

- Domain classes (data entities).
- Service and repository layers.
- REST controllers that expose the microservice endpoints.
- Observer and publisher classes for the eventual synchronization of information or state between services.

Code generation is performed using templates that dynamically insert code extracted from the original monolith stored in the data structure, ensuring the preservation of existing functional logic. Each microservice is encapsulated in a project, ready to be deployed independently as a REST API, maintaining modularity and autonomy when deployed as services.

3.5 Storage and Export of Generated Microservices

Finally, the set of generated microservices is stored in a directory that can be directly imported into a development environment such as Eclipse, IntelliJ, or Visual Studio Code.

4 Software tool Implementation

The tool described in this section, called "MonoDbToMicroDb," implements the proposed method in the Java programming language. Its design integrates static analysis, construction of syntax trees with ANTLR4, and automated generation of artifacts from a data structure that stores the representation of a microservice and its database.

4.1 Tool Design

The developed tool was designed using a modular architecture focused on extensibility and functional isolation of each stage of the decomposition process. Its structure is organized into four core components. The first module handles the preprocessing of the monolith, exploring the physical structure of the original system, recursively traversing the directory tree, and selecting files with the .java extension. The second module implements lexical-syntactic analysis, based on ANTLR4, which constructs the Abstract Syntax Tree (AST) and generates the complete set of rules, tokens, and nodes associated with the Java language. The third module performs structural-semantic analysis, using the Visitor pattern, which traverses the AST to identify structural relationships between classes, dependencies between methods, interaction patterns, and data model manipulations. Finally, the fourth module

executes the synthesis and automated generation of microservices, using the information stored in the data structure to produce the set of derived microservices, their databases, SQL scripts, and REST controllers.

4.2 Configuration Tool

The tool was implemented in Java using the Spring Boot framework and managed through Apache Maven, enabling automated dependency management, project compilation, and reproducible builds. ANTLR4 was integrated as the core-parsing infrastructure, allowing the automatic generation of lexers, parsers, and traversal components from the language grammars. Additional libraries were incorporated for artifact generation and automated testing, facilitating the systematic construction and validation of the decomposition process. This configuration ensures consistency between the language grammars, the parsing process, and the generated analytical structures throughout the execution of the method.

4.3 Internal Organization and Grammar Management

To promote conceptual clarity and system reproducibility, the tool adopts a modular organization that separates parsing grammars, semantic analysis components, and generation templates in order to improve maintainability, extensibility, and reproducibility. The ANTLR grammars were adapted from the official Java language specification and extended to capture relevant software artifacts, including annotations, persistence entities, and dependency relationships required for interaction-sequence reconstruction and database decomposition. This organization facilitates the independent evolution of the analytical and generative components without affecting the overall decomposition workflow.

4.4 Lexer and Parser Generation Process

During the project build phase, Maven automatically runs the ANTLR4 plugin when configuring the XML file. This generates the lexer, parser, base classes (BaseListener, BaseVisitor), and specific visitors needed to traverse the AST. The generated artifacts are stored in the path target/generated-sources/antlr4, a directory that is automatically added to the compiler's class path. This allows the semantic analysis module to directly use the nodes and structures resulting from the syntactic processing, ensuring consistency between the grammar and the parser throughout the entire decomposition process.

4.5 Reproducibility, Extensibility, and Validation

The tool ensures reproducibility using grammars obtained from ANTLR grammars-v4 GitHub repository, explicitly declared dependencies, and a deterministic AST construction process. Its modular design allows any component to be easily replaced or extended, for example, by adopting a more recent language grammar or incorporating additional generation templates for other frameworks.

For testing purposes, an initial synthetic case study was developed, to which unit tests and test designs were applied. The validation of the proposed method followed a mixed approach combining systematic observation, software-testing techniques, and empirical evaluation using the IATC a lightweight coupling indicator.

In the first phase, black-box validation was performed to confirm the correct generation of the derived databases, verifying that the table structures produced by the tool accurately matched the data models identified for each microservice. Subsequently, integration testing and observational analysis were conducted to ensure consistent interaction between the generated microservices and their respective databases, comparing their behavior with that of the original monolithic system.

Finally, an experimental evaluation was conducted using the IATC indicator on both the real-world case study and multiple monolithic systems collected from public Git repositories. The obtained results were analyzed through descriptive statistics and boxplot-based visualizations to determine whether the proposed decomposition effectively reduces structural coupling compared with the original monolithic architectures.

5 Real-world case study

The monolithic system under study is a sales management system that integrates functionalities for managing categories, products, customers, orders, and their corresponding details through the CategoryController, ProductController, ClientController, OrderController, and OrderDetailController controllers. These functionalities support product catalog management, customer

registration and maintenance, and order processing with their associated details. Such operations create interactions between classes and the data model, whose dependencies are illustrated in the class diagram shown in Figure 3.

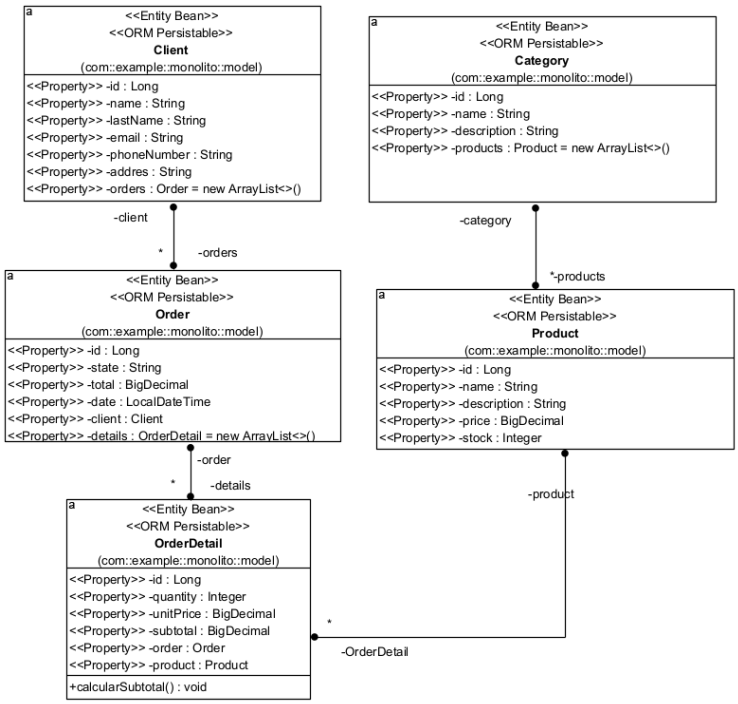

Fig. 3. Class diagram of the sales management system used as a case study.

The class diagram in Figure 3 shows the relationships between the data models. Based on the code analysis, the following relationships between the use cases (controllers) and the data model were obtained:

- CategoryController → model Category.
- ClientController → model Client
- OrderDetailController → model OrderDetail
- OrderController → model Order
- ProductController → model Product

These classes represent business capabilities that interact with entities in the data model (and, therefore, tables in the database). Reviewing the code yielded the relationships shown in Table 2.

Table 2. Relationships between Tables of the Sales Management System

Model	Relations	Annotations
Order	Client	@ManyToOne
Client	OrderDetail	@OneToMany
OrderDetail	Product	@ManyToOne
Product	Category	@ManyToOne

To support the empirical evaluation of the proposed method, a lightweight coupling indicator named Shared Table Coupling Index (IATC) was defined to estimate the degree of shared-data structural coupling before and after decomposition. The indicator provides a simple and interpretable estimation of how strongly system components (use cases or microservices) depend on shared database tables. Lower values indicate reduced shared-data coupling and greater database autonomy among services. The indicator estimates coupling by comparing the total number of database-table accesses performed by the set of use cases or microservices

against the total number of tables in the system, normalized according to the number of analyzed modules and tables. The corresponding formulation is presented in Equation (1).

$$IATC = \frac{accTab - NUT}{|NCU - (NUT * NCU)|} \quad (1)$$

Where:

- IATC: Index for accessing shared tables.
- accTab denotes the total number of unique accesses to database tables.
- NUT corresponds to the total number of tables.
- NCU represents the total number of use cases.

Interpretation of IATC Values:
An IATC value of 0 represents the minimum coupling level, whereas values greater than 0 indicate increasing levels of coupling caused by shared tables. An IATC value of 0 implies that there are no shared tables in the system, meaning that each table is accessed by at most one component. This represents the desired state in a microservices architecture, where each service has full ownership of its data. The following results were obtained when evaluating the system before applying the method and are shown in Table 3.

Table 3. Font Relationships between Tables of the Sales Management System

Business Context	Category	Product	Order	Detail	Client
CategoryController	X				
ProductController	X	X			
OrderController			X	X	X
DetailController		X	X	X	
ClientController			X		X

Table 3 shows that the total number of use cases is five, the total number of table accesses is 11, and the total number of tables in the system is five. Therefore, by applying the IATC formula defined in Equation (2) to the monolithic system, the following result was obtained:

$$IATC = \frac{11 - 5}{|5 - (5 * 5)|} \quad (2)$$

IATC = 0.3, it is demonstrated that the coupling index of the original legacy system is 0.3. Applying the decomposition method to the original legacy system's database, the microservices table accesses resulted in the following and are shown in Table 4:

Table 4. Relationships between Tables of the Sales Management System

Business Context	Category	Product	Order	Detail	Client
CategoryController	X				
ProductController		X			
OrderController			X		
DetailController				X	
ClientController					X

Applying the IATC formula to Equation (3) for the microservices system yielded the following result:

$$IATC = \frac{5 - 5}{|5 - (5 * 5)|} \quad (3)$$

Therefore, the IATC value for the microservices system is 0, indicating the absence of structural coupling caused by shared tables.

6 Complementary Experimental Evaluation

In addition to the real-world case study presented previously, a complementary experimental evaluation was conducted to analyze the applicability of the proposed method across multiple monolithic systems obtained from public repositories. This phase aimed to determine whether the structural coupling reduction observed in the real system could also be identified in other software systems.

6.1 Research Question and Hypotheses

Research Question:
Does a decomposition method that assigns an independent database to each generated microservice significantly reduce the indirect coupling caused by sharing a global database in monolithic systems?

Statistical Formulation:

- Null Hypothesis (H_0): The implementation of a migration method from monolithic systems to microservices, which includes the decomposition of the global database, does not reduce the coupling among the microservices corresponding to the original monolithic system (i.e., the difference between coupling before and after migration is null or not favorable).

- Alternative Hypothesis (H_1): The implementation of a migration method from monolithic systems to microservices, which includes the decomposition of the global database, reduces the coupling among the microservices corresponding to the original monolithic system (i.e., the difference is positive in favor of coupling reduction).

To test the proposed hypotheses, a paired evaluation was conducted, where each project was analyzed independently using its corresponding use cases before and after applying the proposed method.

6.2 Experiment Execution

The experiment was conducted on a population of 20 monolithic systems in order to validate the proposed hypotheses and answer the research question. A population of N = 20 Java monolithic repositories with JPA/Hibernate and database access was established. These systems were selected for the study from public repositories according to availability.

The analyzed repositories correspond to business-oriented Java monolithic applications from different domains, including banking, payment processing, hotel reservation, educational management, restaurant services, and customer account management. All selected systems implement persistence through JPA/Hibernate and follow an object-relational mapping (ORM) approach, which is required by the proposed analysis tool to identify entities, relationships, and database access patterns. Therefore, the current evaluation is representative of Java-based monolithic systems that adopt ORM-based persistence mechanisms, although the results cannot be directly generalized to systems developed with alternative persistence technologies.

6.3 Sample Size Calculation

To calculate the sample size, the formula proposed by (Cochran, 1977) was used, as defined in Equation (4), for a proportion (conservative choice p = 0.5), confidence level of 95% (Z = 1.96), and allowable error E = 0.10 (10%):

$$n_0 = \frac{\left(Z^2 * p * (1 - p)\right)}{E^2} = \frac{\left(1.96^2 * 0.5 * (0.5)\right)}{0.10} = 96.04 \quad (4)$$

Since the population is finite (N = 20), the finite population correction was applied as described in the Equation 5:

$$n = \frac{n_0}{1 + \left(\frac{(n_0 - 1)}{N}\right)} = \frac{96.04}{1 + \left(\frac{(95.04)}{20}\right)} = 16.70 \rightarrow n = 17 \quad \textbf{(5)}$$

Result: with N = 20, p = 0.5, E = 0.10, and a 95% confidence level, a sample of 17 repositories was obtained (rounded up).

6.4 Proportional Allocation by Strata

Stratification was performed dividing the sample into homogeneous groups according to the number of use cases in each system (small, medium, and large). This strategy improves the precision of the estimates and ensures that each stratum is adequately represented in the analysis, following the methodological principles of stratified sampling described by (Cochran, 1977).

Rationale for Stratification:
The systems differ in size and complexity. Therefore, the number of use cases was selected as an operational indicator for grouping systems into small, medium, and large categories. Use cases were identified through public controller classes or detected REST endpoints/entry points. This indicator is relevant because it influences the number of interaction sequences and the potential for data coupling across repositories.

Proportional Allocation:
Based on the population of 20 repositories, the proportional allocation for the final sample of 17 repositories resulted in the following strata distribution:

- Small: CU ≤ 5 (7 systems)
- Medium: ≤ CU ≤ 15 (6 systems)
- Large: CU > 15 (4 systems)

The size-based strata (use cases per system) are defined in Equation (6):

$$n_h = n * \left(\frac{N_h}{N}\right) \quad \textbf{(6)}$$

Where:

- n_h = resulting stratum size.
- n = 17 = sample size.
- N_h = number of small-sized, medium-sized, and large-sized cases in the population.
- N = 20 = population size.

n_small = 17 × (8 / 20) = 6.8 → rounded up to 7
n_medium = 17 × (7 / 20) = 5.95 → rounded up to 6
n_large = 17 × (5 / 20) = 4.25 → rounded down to 4

Thus, n = 7 + 6 + 4 = 17.

Final Selection: Within each stratum, the corresponding n_small / n_medium / n_large repositories were randomly selected (without replacement). These repositories are presented in Table 5.

Table 5. Repositories divided into strata according to their use cases

Id	Repository	Use Cases	Stratum
1	building-modular-monoliths-using-spring	18	Large
2	modular-monolith-restaurant	20	Large
3	petclinic	22	Large
4	bookdelivery	27	Large
5	monopay-wallet	6	Medium
6	modular-monoliths	7	Medium
7	ddd-modular-monolith-customer-account	8	Medium
8	simplebanking	9	Medium
9	rolepermissionexample	11	Medium
10	brewery-monolith	12	Medium
11	hotel-reservation-system	4	Small
12	kiosk-queue-management-system	2	Small
13	wind-turbines-edge-ai	3	Small
14	micro-server	3	Small
15	educational-platform	4	Small
16	blog-modular-monoliths-across	4	Small
17	microservices	5	Small

Although the sample size is constrained by the availability of public repositories satisfying the selection criteria, the stratified sampling strategy ensured the inclusion of systems with different sizes and complexity levels. The final sample contains small, medium, and large systems, reducing the risk that the evaluation is biased toward a particular project size and improving the representativeness of the experimental results.

6.5 Application of the IATC Coupling Indicator

Once the number of repositories and their corresponding strata had been defined, the IATC coupling indicator was applied to measure coupling before and after applying the migration method. The obtained results were recorded in a structured data matrix, allowing a systematic comparison between the initial state (monolith) and the resulting state (microservices), in order to determine whether coupling was reduced after applying the decomposition method. Table 6 presents the collected data organized for subsequent analysis and interpretation of results.

Table 6. Repositories divided into strata according to their use cases

Id	Repository	Use Cases	Monolith IATC	Migrated IATC	Coupling Reduction (%)	Stratum
1	building-modular-monoliths-using-spring	18	0.82	0.00	100	Large
2	modular-monolith-restaurant	20	0.68	0.22	68	Large
3	petclinic	22	0.80	0.33	59	Large
4	bookdelivery	27	0.84	0.00	100	Large
5	monopay-wallet	6	0.58	0.01	98	Medium
6	modular-monoliths	7	0.55	0.00	100	Medium

Id	Repository	Use Cases	Monolith IATC	Migrated IATC	Coupling Reduction (%)	Stratum
7	ddd-modular-monolith-customer-account	8	0.47	0.00	100	Medium
8	simplebanking	9	0.80	0.00	100	Medium
9	rolepermissionexample	11	0.57	0.00	100	Medium
10	brewery-monolith	12	0.63	0.24	62	Medium
11	hotel-reservation-system	4	0.60	0.16	73	Small
12	kiosk-queue-management-system	2	0.40	0.00	100	Small
13	wind-turbines-edge-ai	3	0.65	0.00	100	Small
14	micro-server	3	0.46	0.00	100	Small
15	educational-platform	4	0.72	0.00	100	Small
16	blog-modular-monoliths-across	4	0.50	0.00	100	Small
17	EmployeeManagementSystem	5	0.49	0.00	100	Small

To evaluate whether the proposed method produces a significant improvement (coupling reduction) compared with the original monolithic system, it is necessary to compare paired measurements taken from the same experimental units. Since the observations are paired, the appropriate statistical approach is to analyze the individual differences between both conditions.

6.6 Descriptive and Statistical Analysis

To evaluate the effectiveness of the proposed method, the Shared Table Coupling Index (IATC) was computed before and after migration for each repository. The results show a consistent reduction in coupling across all evaluated systems. On average, the IATC decreased from 0.621 in the monolithic systems to 0.056 after migration, yielding a mean reduction of 0.565. This substantial decrease indicates that the proposed method effectively reduces indirect coupling caused by shared database tables.

Descriptive statistics further support this observation. The median IATC value decreased from 0.600 in the monolithic systems to 0.000 after migration, indicating that in at least half of the evaluated cases, coupling was eliminated. Additionally, the standard deviation of the differences (0.148) suggests a relatively consistent reduction across repositories, despite variations in system size and complexity.

To assess the statistical significance of the observed reduction, the normality of the difference vector was evaluated using the (Shapiro & Wilk, 1965) test. The result ($W=0.873$, $p=0.025$) indicates that the normality assumption is not satisfied. Therefore, a non-parametric approach was adopted.

A (Wilcoxon, 1945) signed-rank test was performed to compare the paired observations, yielding a statistically significant result ($W = 153.0$, $p < 0.001$). This confirms that the reduction in IATC after applying the proposed method is not due to random variation.

To quantify the magnitude of the effect, Cohen's d for paired samples was computed and it is presented in equation 7:

$$d = \frac{\bar{d}}{s_d} \tag{7}$$

The resulting value ($d = 3.81$) indicates an extremely large effect size**,** far exceeding the conventional threshold of 0.8. This demonstrates that the reduction in coupling is not only statistically significant but also practically meaningful.

6.7 Visual Analysis of Results

In addition to the statistical analysis, a visual exploration of the results was conducted to better understand the behavior of coupling reduction across different system sizes. The generated visualizations clearly illustrate the reduction in coupling (IATC) across the evaluated repositories before and after applying the proposed method.

Figure 4 presents a paired comparison for each repository, where each line connects the initial (monolithic) value to the corresponding post-migration value. This representation highlights the reduction on a per-system basis, allowing a direct observation of the improvement achieved in each case.

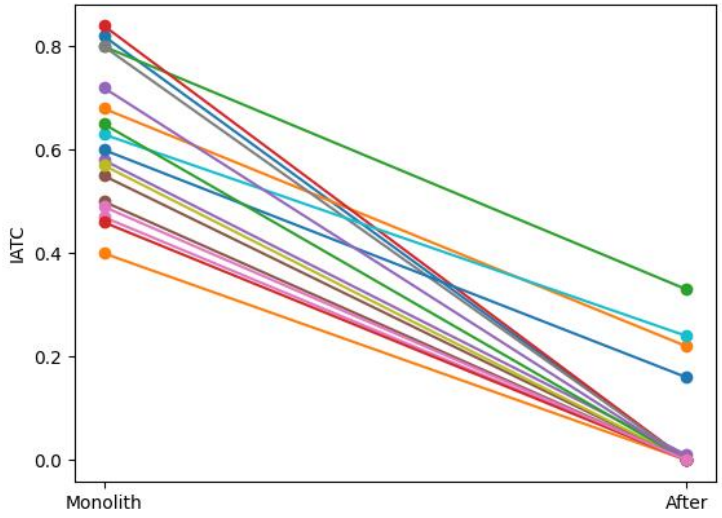

Fig. 4. Paired reduction of IATC per repository.

Figure 5 shows a stratified boxplot that groups the results by system size (small, medium, and large). This visualization reveals how system size influences the magnitude and variability of the reduction, while also demonstrating that the proposed method consistently reduces coupling across all groups.

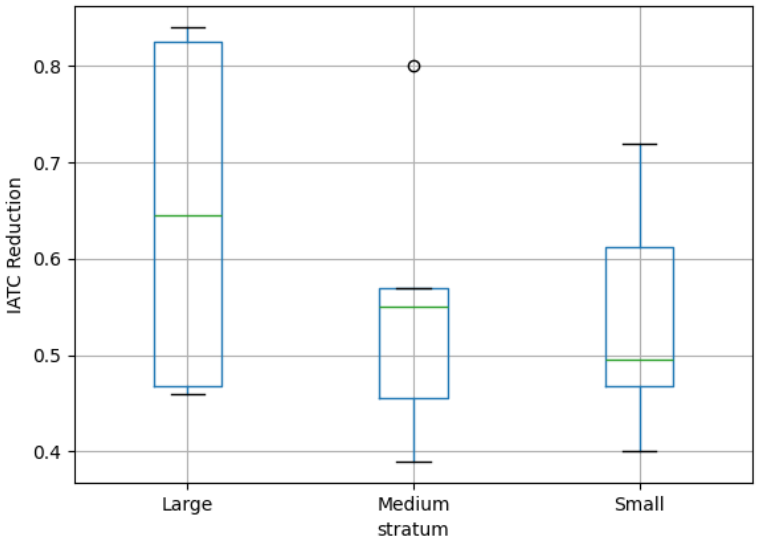

Fig. 5. IATC reduction by system size.

These visual results complement the statistical analysis by providing an intuitive representation of the coupling reduction observed across repositories and system sizes.

6.8 Threats to Validity

The proposed evaluation is subject to several threats to validity. Regarding construct validity, the IATC coupling indicator was used only as a lightweight comparative estimation of shared-data coupling and does not represent all possible forms of software dependency, such as runtime or semantic coupling.

Concerning internal validity, the proposed method relies exclusively on static analysis; therefore, dynamically generated dependencies or runtime behaviors may not be fully identified. To mitigate this limitation, the evaluation focused on enterprise monolithic systems with explicit controller-service-repository structures.

Regarding external validity, the analyzed repositories correspond mainly to Java monolithic systems using ORM JPA/Hibernate and REST-based architectures. Furthermore, although the proposed method is conceptually extensible to multiple programming languages through ANTLR grammars, the current implementation and experimental validation were conducted exclusively for Java-based systems. Consequently, the obtained results may not directly generalize to other technological ecosystems or programming languages.

Finally, conclusion validity was addressed by applying paired statistical analysis and adopting a non-parametric Wilcoxon signed-rank test after verifying the non-normality of the data distribution.

7 Results Analysis

The obtained results indicate that the proposed decomposition method successfully separates the data model associated with each identified microservice, reducing shared-data dependencies during migration toward a microservices architecture. Lower IATC values represent lower levels of shared-data coupling, whereas higher values indicate stronger dependencies between services and shared database structures.

The combined statistical and visual analyses provide empirical evidence supporting the applicability of the proposed method for reducing shared-data coupling in monolithic systems.

First, the consistent reduction observed across the evaluated repositories suggests that the method systematically mitigates shared database dependencies, which are a major source of indirect coupling in monolithic architectures. The mean IATC decreased from 0.621 in the monolithic systems to 0.056 after migration, corresponding to an average reduction of approximately 91%. Furthermore, the median value decreased from 0.600 to 0.000 after migration, indicating that in at least half of the evaluated systems the shared-data coupling was eliminated.

The observed reduction in coupling is achieved through the database-per-service principle and the controlled replication of shared entities. Consequently, the resulting architecture introduces trade-offs commonly associated with microservices, including data duplication, eventual consistency requirements, and additional synchronization mechanisms between services. These trade-offs are consistent with established microservice design practices and are accepted in exchange for improved service autonomy and reduced dependence on a shared database. The present study focuses on evaluating coupling reduction through the IATC indicator; therefore, the impact of replication and synchronization mechanisms on system performance remains a topic for future work.

A threat to validity is that the proposed tool operates under a set of structural assumptions regarding the analyzed systems. Therefore, some functionalities present in the evaluated repositories could not be automatically analyzed when the required source-code and persistence patterns were not available. Although this limitation may affect the completeness of the generated microservice candidates, it does not affect the coupling measurements reported for the candidates successfully identified by the method.

Second, the statistical analysis indicates that the observed reduction is statistically significant. Since the Shapiro–Wilk test rejected the normality assumption for the paired differences ($W = 0.873$, $p = 0.025$), a non-parametric Wilcoxon signed-rank test was applied. The obtained result ($W = 153.0$, $p < 0.001$) indicates that the reduction in coupling observed after applying the proposed method is unlikely to be explained by random variation alone.

Additionally, the visual analysis of the repositories reveals a consistent reduction trend across systems of different sizes. Although larger systems exhibit greater variability due to their increased structural complexity, the overall reduction pattern remains stable. Smaller systems, despite initially presenting lower coupling values, also benefited from the proposed method by achieving consistent reductions in shared-data dependencies.

An important observation derived from the experimental evaluation is that a considerable number of repositories achieved an IATC value equal to zero after decomposition. This result indicates the complete elimination of structural coupling caused by shared database tables. Such behavior is consistent with the database-per-service principle, where services operate independently without sharing persistence structures.

However, a smaller subset of systems retained non-zero IATC values after migration. These cases correspond mainly to repositories containing entities or tables with cross-cutting responsibilities that participate in multiple business capabilities or interaction sequences. In such scenarios, the proposed method preserves certain shared data relationships to maintain functional consistency and avoid incorrect fragmentation of highly related domain entities.

Overall, the obtained results provide empirical evidence supporting the applicability of the proposed method for reducing shared-data coupling in monolithic systems migrated toward microservices architectures.

8 Future Work

As future work, several research directions are identified based on the limitations and results obtained from the proposed method. First, the method can be extended to support heterogeneous and multi-language systems, increasing its applicability in industrial environments where legacy monolithic applications are commonly implemented using multiple programming languages and technologies. Although the current implementation was validated using Java-based systems, the use of ANTLR grammars provides a foundation for extending the approach to other programming languages.

Second, future research may explore the incorporation of artificial intelligence techniques to support the decomposition process. In particular, machine learning and graph-based approaches could assist in identifying microservice boundaries, detecting structural dependencies, and suggesting alternative decompositions based on patterns learned from existing software systems. Such techniques could complement the current static-analysis approach and provide additional decision-support capabilities for software architects.

Finally, future work includes the formal analysis of communication and consistency mechanisms between the generated microservices. Since the proposed method identifies interaction relationships and shared entities during decomposition, these relationships could be used to automatically derive communication structures based on domain events. This would enable further verification of consistency constraints and interaction correctness in the resulting microservices architecture.

Acknowledgment

The Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI), supported this research. The authors also gratefully acknowledge the Tecnológico Nacional de México and the Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) for their institutional support throughout the development of this work.

References

Abgaz, Y. M., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G. M., Yilmaz, M., Buckley, J., & Clarke, P. M. (2023). Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering, 49*(8), 4213–4242. https://doi.org/10.1109/TSE.2023.3287297

André, M., Raglianti, M., Serbout, S., Cleve, A., & Lanza, M. (2025). *An empirical study on database usage in microservices* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2510.20582

Andrade, B., Santos, S., & Rito Silva, A. (2023). A comparison of static and dynamic analysis to identify microservices in monolith systems. In B. Tekinerdogan, C. Trubiani, C. Tibermacine, P. Scandurra, & C. E. Cuesta (Eds.), *Software architecture: 17th European Conference, ECSA 2023, Istanbul, Turkey, September 18–22, 2023, proceedings* (Lecture Notes in Computer Science, Vol. 14212, pp. 354–361). Springer. https://doi.org/10.1007/978-3-031-42592-9_25

Michael Ayas, H., Leitner, P., & Hebig, R. (2023). An empirical study of the systemic and technical migration towards microservices. *Empirical Software Engineering, 28*(4), Article 85. https://doi.org/10.1007/s10664-023-10308-9

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Addison-Wesley.

Castillo-Estrada, C. M., Cancino-Villatoro, K., Alvarez-Oval, L. A., Muñoz, M., & de la Cruz Vazquez, A. (2025). Decomposition methodologies of relational databases for migration from systems monolithic to microservices: A systematic literature mapping. In J. Mejía, M. Muñoz, A. Rocha, F. J. Espinosa-Faller, & J. A. Trejo-Sanchez (Eds.), *New challenges in software engineering* (Studies in Computational Intelligence, Vol. 1209, pp. 183–205). Springer. https://doi.org/10.1007/978-3-031-90310-6_13

Cochran, W. G. (1977). *Sampling techniques* (3rd ed.). Wiley.

Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

Filippone, G., Mehmood, N. Q., Autili, M., Rossi, F., & Tivoli, M. (2023). 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)* (pp. 47–57). IEEE. https://doi.org/10.1109/ICSA56044.2023.00013

Freitas, F., Ferreira, A. L., & Cunha, J. (2023). A methodology for refactoring ORM-based monolithic web applications into microservices. *Journal of Computer Languages, 75*, Article 101205. https://doi.org/10.1016/j.cola.2023.101205

Hassan, H., Abdel-Fattah, M. A., & Mohamed, W. (2024). Migrating from monolithic to microservice architectures: A systematic literature review. *International Journal of Advanced Computer Science and Applications, 15*(10), 104–116. https://doi.org/10.14569/IJACSA.2024.0151013

Hao, J., Zhao, J., & Li, Y. (2023). Research on decompostion method of relational database oriented to microservice refactoring. In *2023 24th Asia-Pacific Network Operations and Management Symposium (APNOMS)* (pp. 282–285). IEEE. https://doi.org/10.34385/proc.75.PS2-06

Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R., & Zheng, Q. (2021). Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering, 47*(5), 987–1007. https://doi.org/10.1109/TSE.2019.2910531

Kalia, A. K., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., & Banerjee, D. (2021). Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1214–1224). Association for Computing Machinery. https://doi.org/10.1145/3468264.3473915

Kimmel, P. (2008). *Manual de UML*. McGraw-Hill Interamericana.

Krause, A., Zirkelbach, C., Hasselbring, W., Lenga, S., & Kröger, D. (2020). Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)* (pp. 9–16). IEEE. https://doi.org/10.1109/ICSA-C50368.2020.00011

Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment, 14*(13), 3348–3361. https://doi.org/10.14778/3484224.3484232

Lewis, J., & Fowler, M. (2014, March 25). *Microservices: A definition of this new architectural term*. Martin Fowler. https://martinfowler.com/articles/microservices.html

Maharjan, R., Sooksatra, K., Cerny, T., Rajbhandari, Y., & Shrestha, S. (2025). A case study on monolith to microservices decomposition with variational autoencoder-based graph neural network. *Future Internet, 17*(7), Article 303. https://doi.org/10.3390/fi17070303

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In *2017 IEEE 24th International Conference on Web Services (ICWS)* (pp. 524–531). IEEE. https://doi.org/10.1109/ICWS.2017.61

Newman, S. (2015). *Building microservices: Designing fine-grained systems*. O'Reilly Media.

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Richardson, C. (2018). *Microservices patterns: With examples in Java*. Manning.

Richardson, C. (n.d.). *Pattern: Database per service*. Microservices.io. Retrieved June 22, 2026, from https://microservices.io/patterns/data/database-per-service.html

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika, 52*(3/4), 591–611. https://doi.org/10.2307/2333709

Volynsky, E., Mehmed, M., & Krusche, S. (2022). Architect: A framework for the migration to microservices. In M. H. Miraz, G. Southal, M. Ali, & A. Ware (Eds.), *2022 International Conference on Computing, Electronics and Communications Engineering (iCCECE)* (pp. 71–76). IEEE. https://doi.org/10.1109/iCCECE55162.2022.9875096

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin, 1*(6), 80–83. https://doi.org/10.2307/3001968