Micro4Delphi: A Process for the Modernization of Legacy Systems in Delphi to Microservice Architecture

Lucas Fernando Fávero[1,2], Gabriel Soares Mário[1,3] and Frank José Affonso[1][a]
[1]*Department of Statistics, Applied Mathematics and Computation, São Paulo State University – UNESP, PO Box 178, Rio Claro, 13506-900, São Paulo, Brazil*
[2]*Integrativa, Chile Street, 240, Catanduva, 15800-430, São Paulo, Brazil*
[3]*SuperSoft, 16A Avenue, 1190, Rio Claro, 13506-720, São Paulo, Brazil*

Keywords: Software Modernization, Delphi, Legacy Systems, Microservice Architecture, MSA.

Abstract: The modernization of legacy systems to microservice architecture (MSA) has been a subject of interest in both academic and industrial areas. This architectural style has facilitated the development of software systems by composing them as a collection of small and loosely coupled services, each running in its process and communicating with lightweight mechanisms. In parallel, Delphi is an integrated development environment (IDE), based on the Object Pascal programming language, that enables the rapid application development of software for desktop, mobile, web, and console applications. Although the software systems developed in Delphi have considerable relevance in contemporary software, there is a lack of documented processes that facilitate the modernization of legacy systems in Delphi to MSA. This paper presents the Micro4Dephi, a modernization process based on six well-defined activities. Each activity is constituted by a step set, which may vary in number and content, thus allowing such activities to be performed. A case study was conducted to show the applicability of the process proposed in this paper. The results provide important evidence that enables a clear perspective on the process's contribution to software modernization.

1 INTRODUCTION

Microservice architecture (MSA) has emerged as a feasible alternative for the design or modernization of legacy or monolithic systems for computational scenarios that are more modern (e.g., elastic) and enable the absorption of new users or execution environment needs. Such systems must be divided into small and loosely coupled services, each running in isolated processes and communicating via lightweight mechanisms. Furthermore, it is suggested that these services be designed around the business capabilities of these systems and must be deployable independently through automated processes (Lewis and Fowler, 2019; Newman, 2021).

Drawing a parallel between legacy and monolithic systems, Pressman and Maxim (2019) suggest that legacy can be understood as a system developed with obsolete computing resources (e.g., architectural models, programming languages, databases) for the current computing scenario. According to (Lewis and Fowler, 2019), the absence of developers also becomes a system developed with current technologies a legacy, as the developers must utilize techniques to comprehend it for any maintenance activity. As argued by Dragoni et al. (2017), monolithic systems were designed with an architectural organization that presents challenges in terms of scalability and adaptability to evolving requirements. Examples of adversities associated with monolithic systems include high coupling, concurrency in teams working on the same code base, and the high impact of changes Newman (2021). Therefore, regardless of the specific type of software (i.e., legacy or monolithic), software modernization has been revealed as a feasible alternative for companies that intend to overcome the aforementioned adversities and enhance the availability of their systems to end users, considering recent technologies, architectural models, and modern computing scenarios (Soldani et al., 2018).

Regardless of the system's organizational structure, whether monolithic or not, the focus of this paper is on the modernization of legacy systems developed in Delphi for MSA. According to Embarcadero (2024), Delphi is an integrated development environ-

[a] https://orcid.org/0000-0002-5784-6248

Fávero, L. F., Mário, G. S. and Affonso, F. J.
Micro4Delphi: A Process for the Modernization of Legacy Systems in Delphi to Microservice Architecture.
DOI: 10.5220/0013434400003929

In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 2*, pages 328-339
ISBN: 978-989-758-749-8; ISSN: 2184-4992

ment (IDE), supported by the Object Pascal programming language, that enables the rapid application development of software for desktop, mobile, web, and console applications. The decision to modernize systems developed in Delphi was predicated upon several factors, including its sustained importance within the current technological environment. This relevance can be attributed to the IDE's robust developer community and extensive development resources, such as rapid development and seamless integration with current technologies (TIOBE, 2024).

As revealed by the investigation conducted by Almeida et al. (2024) and Fávero and Affonso (2024), software modernization initiatives identified in the existing literature can be viewed as partial solutions, as such initiatives address specific steps within a complex and wide-ranging context. Therefore, it can be stated that there is no modernization process in the literature that provides a solid foundation for supporting the transition of legacy systems from Delphi to MSA, encompassing activities from understanding the legacy system to monitoring microservices in the execution environment. Aiming to contribute to the presented research scenario, this paper presents Micro4Delphi, a process structured in six activities covering the spectrum of actuation, from legacy system comprehension to microservice monitoring. It is important to highlight that each process's activity has a set of steps that can vary in number and purpose to achieve the desired outcome.

The findings reveal that Micro4Delphi has a good potential to contribute to software modernization, as it is the first initiative that addresses the modernization of systems developed in Delphi to MSA. These findings also suggest that Micro4Delphi represents a theoretical framework by which this process may be instantiated to support other programming languages. Finally, it is important to highlight that the case study detailed in this paper provides a significant theoretical and practical contribution to the advancement of microservice application development using Delphi.

The reminder of this paper is organized as follows: Section 2 presents the background and related work; a description of Micro4Delphi is provided in Section 3; Section 4 presents a case study to show the applicability of Micro4Delphi; and Section 5 summarizes the conclusions and perspectives for further research.

2 BACKGROUND AND RELATED WORK

This section presents the background and related work that contributed to the development of Micro4Delphi. Initially, concepts of Delphi, MSA, and software modernization are described. Next, related work on the software modernization of legacy systems to MSA is addressed.

Delphi. According to Embarcadero (2024), Delphi is a robust and contemporary IDE that provides the advantages of a unified code base with the benefits of publishing native applications for any device. In terms of innovations, this IDE also provides support for the plugin architecture of artificial intelligence (e.g., OpenAI[1]). Finally, it is also noteworthy that Delphi has an active community, as evidenced by its ranking on the Tiobe index (TIOBE, 2024). Based on the presented context, the motivation for modernizing software developed in Delphi for MSA appears to be well-founded and compelling.

MSA. Microservice architecture is an architectural style that facilitates the development of applications as a set of small, loosely coupled, and independent microservices (Lewis and Fowler, 2019). Regarding the development and deployment activity, such microservices may be written in different programming languages and use different databases. Because of loose coupling, microservices can be independently deployed by fully automated processes through the utilization of lightweight, container-based platforms (Soldani et al., 2018). Richardson (2018) suggests that a pattern language for microservices can be used to support the development of a new system or the modernization of an existing one. These patterns represent solutions to common knowledge problems that overcome adversities related to, for example, decomposition, transactions, communication via messages, service discovery, among others.

Software Modernization. The monolith was one of the earliest architectural approaches utilized in the software industry because of its ease of development and deployment and inherent benefits (e.g., simplified code base management and direct scalability within a single software unit). However, as monolithic systems have grown in size, many adversities have emerged, including low productivity, extended delivery times, and the overall complexity of the code bases (Soldani et al., 2018). These challenges have impeded the ability to incorporate emerging technologies and potentially reduce development cycles and releases. In this direction, Colanzi et al. (2021a) suggests the modernization of legacy systems to MSA as a feasible alternative to overcome such challenges.

As related work, to the best of our knowledge, there is no process for the modernization of legacy systems developed in Delphi to MSA. In order to ensure the originality of the process proposed in this

[1]https://openai.com

paper, a systematic mapping was conducted based on guidelines established by Petersen et al. (2015). After a systematic process of investigation, 43 studies were selected because they presented some evidence related to software modernization (i.e., legacy or monolithic) to MSA. Because of space and scope limitations, the details of this mapping will not be presented in this paper. The detailed research protocol and the list of primary studies that served as the basis for the modernization process presented in this paper are available in Fávero and Affonso (2024).

3 MICRO4DELPHI

This section presents Micro4Delphi, a process resulting from a collaborative endeavor involving software engineering researchers and practitioners from two companies that specialize in the development of software systems in Delphi.

On the **researchers' side**, a literature mapping was conducted by Fávero and Affonso (2024) and the findings provided concrete evidence regarding a set of macro activities that are essential to a modernization process, namely: planning, analysis, decomposition, development, integration, and monitoring. Moreover, such mapping also identified a sequence of steps that can be followed to achieve the objectives of these activities, generating the initial draft of the process.

On the **practitioners' side**, a rigorous process analysis was conducted to evaluate the relevance and applicability of the outlined activities and steps (i.e., the initial version). Such analysis provided a set of evidence that was used to improve the proposed process (i.e., the second version). It is worth highlighting that a presentation on software modernization and MSA concepts was carried out before the aforementioned analysis. The purpose of this presentation was to establish a unified understanding of this background among the practitioners engaged in this activity.

Based on the evidence provided by the practitioners specialized in Delphi, a meeting for conflict resolution was conducted. The objective of this meeting was to resolve discrepancies and uniform the consensus between researchers and practitioners on the activities and steps for the Micro4Delphi process. As a result, a definitive version of the process was established, as illustrated in Figure 1. Next, a description of each activity and its respective steps is addressed.

As illustrated, the development team (i.e., domain specialists and software architects) conducts the software modernization of the legacy system to the MSA through a cyclical and incremental approach. According to Pressman and Maxim (2019), iterative approaches have proven to be feasible in modernization, enabling not only the translation of legacy systems but also the incorporation of new requirements to fulfill updated user demands. Regarding the legacy system, the proposed process was designed without consideration of the existence of documentation artifacts. Therefore, it is anticipated that researchers and practitioners interested in utilizing the process will have access solely to the source code repository of this system, to transform it into an executable system (see **Legacy System**).

Planning is an indispensable activity in any software development project, including the modernization of legacy systems to MSA. By planning modernization activities, organizations and development teams increase the chances of success by anticipating potential adversities and ensuring that objectives are achieved within the defined deadlines and resources. In short, this activity involves the delineation of the objectives, strategies, and resources necessary to achieve the desired results. From the perspective of the organization, the adoption of a development support methodology and the use of infrastructure to enable distributed application development have been demonstrated to be effective approaches for conducting modernization practices (Auer et al., 2021; Li et al., 2020). Next, the steps associated with the planning activity are described.

1.1.Scope definition. This step is essential for any initiative that aims to support the modernization of legacy systems to MSA, as it involves establishing a precise definition of the scope and goals of the systems (i.e., legacy and modernized). To do so, this step aims to establish a perimeter for the functionalities/operations that must be migrated to microservices, aligned with all requirements that must be met for both legacy and modernized systems. Furthermore, this step must be conducted in a manner that is aligned with the expectations of stakeholders and focused on the successful delivery of the objectives of the modernized system (Li et al., 2020).

1.2.Team structuring. This step proposes a reorganization of the teams responsible for modernization according to specific criteria, including size, multidisciplinary knowledge capacity, independence, autonomy, and alignment with DevOps (Development and Operations) practices. The delineation of responsibilities, formation of a multidisciplinary team, establishment of leadership, promotion of collaboration and communication, and fostering of continuous learning can facilitate the organization of effective teams (Mazzara et al., 2021).

1.3.Organizational culture. This step is based on the premise that modernization is an activity that goes

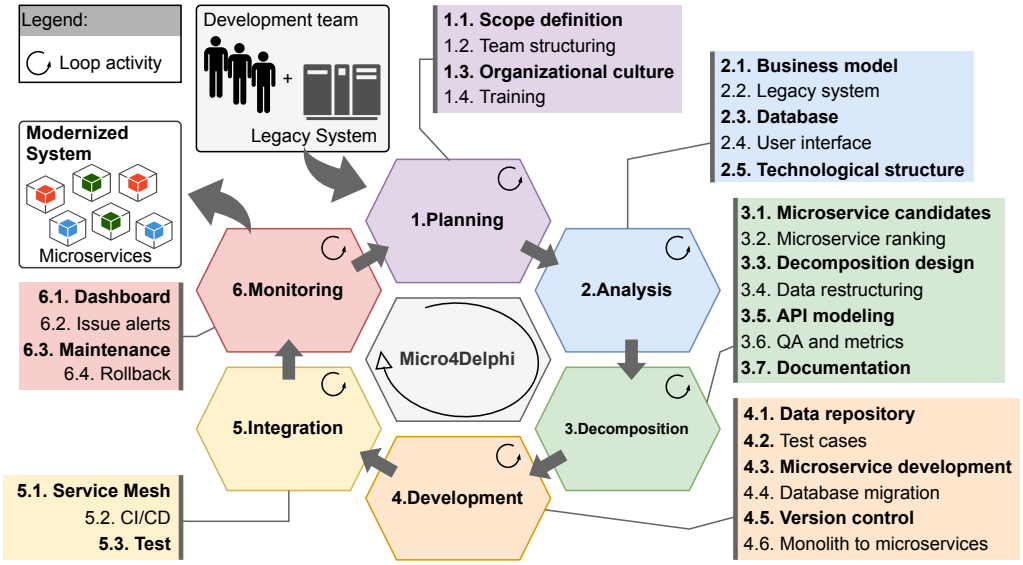

Figure 1: The Micro4Delphi Process.

beyond the translation of a legacy system into MSA, requiring a change in the organizational culture of companies. Such changes require a shift in focus from business capabilities to the continuous and valuable delivery of software. A learning-oriented mindset, encouragement of experimentation, collaboration, open communication, resilience, flexibility, and committed leadership are some factors that can contribute to the organizational culture of companies (Mazzara et al., 2021; Colanzi et al., 2021b).

1.4.Training. This step has proven to be essential for the planning activity in a modernization process. In this direction, experience reports presented by Soldani et al. (2018) suggested that an understanding of the microservices architecture style and training in the technology stack that will be utilized in the target system are factors that can facilitate success in modernizing a legacy system to this architectural style.

Analysis is a crucial activity element in the modernization process, as it enables developers to ascertain the availability of software artifacts from legacy systems, including source code repositories, binary systems, and documentation artifacts. As a result, it is expected that enough information will be gathered to facilitate the formulation of a decision-making process that will determine the optimal point of initiation for the decomposition process. This requires the identification of the legacy system's constituent components and functionalities, an understanding of its business domain boundaries, an analysis of the dependencies between the aforementioned components, and the establishment of a preliminary foundation for the design and implementation of the MSA (Krause et al., 2020; Ma et al., 2022b; Bandara and Perera, 2020). Next, a description of the steps associated with the analysis is addressed.

2.1.Business model. This step suggests that the development team should have a comprehensive understanding of the business operations and processes implemented in the legacy system. As a result, this investigation provides valuable insights to development teams on how IT systems can best support and leverage the organization's business objectives and operations. To do so, developers must investigate how the different parts of the legacy system support or affect the organization's business processes. This investigation enables the system boundaries and interactions to be defined, which can then be redefined or optimized in the new architecture (Dehghani et al., 2022).

2.2 to 2.4 (Legacy systems, database, and user interface). To ensure a comprehensive understanding of the legacy system, the development team must conduct an architectural evaluation in this step, covering internal and external components, functionalities, integrations, database, and user interface. Here, the objective is to identify problematic points, such as obsolete code, complex dependencies, performance bottlenecks, and areas of low scalability. This analysis facilitates the team's understanding of modernization challenges and risks, enabling the formulation of a suitable strategy to ensure the modernized system aligns with company requirements (Trabelsi et al., 2022). As evidenced by Lewis and Fowler (2019), a microservice can be defined as a self-contained application that encompasses code for its functionality, data storage, and user interface. Therefore, when addressing limitations from the perspective of functionality and database, it is imperative to consider

how users interact with the aforementioned functionality (Ma et al., 2022b). This involves gathering feedback on usability, user experience, visual design, and other aspects related to interacting with the software. Based on such insights, the development team can design and implement a modern, intuitive, and effective user interface for the microservices, ensuring a positive experience for end users (Prasandy et al., 2020).

2.5.Technological structure. This step concerns the definition of the software architecture and the technologies that will be adopted for the microservice implementation. This includes selecting the programming languages, frameworks, platforms, and infrastructure services to support the new architectural design. The technology structure should be chosen based on the business requirements, the features of the legacy system, the need for scalability, availability, and security, and the development team's capabilities. Effective planning at this step can facilitate a seamless transition to the MSA and the success of the modernization project (Michael Ayas et al., 2021).

Decomposition is an activity that focuses on software architecture, with special attention on the extraction of microservice candidates from the legacy system. According to Lewis and Fowler (2019), the design of microservices must consider the organizational and business aspects that they offer to users. The challenge of this activity is to determine the optimal size for the microservices, which should be divided into smaller units with low coupling and high cohesion. Once designed, each microservice should be responsible for a delimited context, providing a cohesive set of functionalities with a well-defined scope to meet a specific business capability (Krause et al., 2020). Although there are (semi)automated techniques that facilitate the decomposition of systems into microservices, it is recommended that, initially, functionalities with low impact or risk be prioritized during a modernization process (Osman et al., 2022). To do so, the development team must establish a strategy for selecting candidate microservices that will add greater value and have fewer external dependencies. Next, the steps associated with the planning activity are described.

3.1.Microservice candidates. During this step, a rigorous analysis of legacy system functionalities is essential to determine the viability of potential microservice candidates (Sellami et al., 2022). Among the approaches existing in the literature that can support decomposition, the most frequently utilized are: (i) decomposition by delimited context, which aims to identify the areas of the system that have cohesive and well-defined responsibilities, representing a delimited business context; (ii) adequate granularity, which aims to evaluate the size and complexity of the functionalities of the legacy systems. Microservices that are too small can result in excessive communication, while those that are too large can compromise the scalability and maintainability of the modernized system; and (iii) loose coupling, which aims to identify functionalities with few external dependencies that can be isolated and maintained independently. This approach helps to minimize coupling between microservices and facilitates the evolution and maintenance of the modernized system.

3.2.Microservice ranking. In this step, candidate microservices are ranked in order of importance, aiming to promote continuous and valuable software integrity (Sellami et al., 2022). In this sense, the investigation conducted by Fávero and Affonso (2024) suggests the following guidelines: (i) the most critical or essential use cases for the business; (ii) the microservices that offer the greatest commercial value or that address the most urgent customer needs; (iii) the microservices that are prerequisites for other microservices or that have few external dependencies; and (iv) the microservices that are simpler to implement or that have lower technical risk. It is also noteworthy that scoring and classification strategies, such as value versus effort analysis, have proven to be a feasible alternative in other software domains and can be easily adapted to the context of this paper.

3.3.Decomposition design. At this step, developers should focus on establishing the communication interface for each microservice, respecting the system's business capabilities. In this sense, they should also dedicate particular attention to the management of dependencies between microservices, aiming to minimize coupling and ensure independence. To do so, it is recommended that architectural patterns and good software engineering practices be used to design and implement microservices, such as containers, container orchestration, API Gateway, and Circuit Breaker (Richardson, 2018). Moreover, robust testing and validation strategies have also proven a feasible alternative for the decomposition and design of new microservices, thereby ensuring the quality and integrity of these services (Trabelsi et al., 2022).

3.4.Data restructuring. Similarly to the previous step, the developers must restructure (i.e., decompose) the legacy system's databases in this step to make them compatible with the microservices that will be developed. To do so, they should focus on identifying the different types of data, relationships between entities, and coupling of database entities. This will facilitate the decomposition of the legacy system's data model into smaller, more specialized schemas, which can then be tailored to meet the spe-

cific requirements of each microservice. Moreover, it is recommended that this activity be carried out in parallel with the previous one (Parikh et al., 2022).

3.5.API modeling. In this step, developers should focus their efforts on modeling the APIs, which represent the microservices contracts (Kyryk et al., 2022). This step can be summarized as follows (Erl et al., 2012): (i) identification of the resources or functionalities (i.e., business entities) that will be exposed by the microservices; (ii) definition of RESTful endpoints, ensuring that each resource is represented by a unique URI (Uniform Resource Identifier) and accessible through standard HTTP methods, such as GET, POST, PUT, and DELETE; (iii) selection of appropriate data formats to represent the API resources (e.g., JSON - JavaScript Object Notation); (iv) documentation of the API contracts clearly and comprehensively, specifying the input parameters, expected responses, and possible HTTP status codes; (v) versioning the API to ensure compatibility with future versions; (vi) security-oriented design so that authentication and authorization mechanisms are implemented in such APIs so that the system resources and data be protected; (vii) API documentation to facilitate discovery and understanding of its functionality; and (viii) testing (e.g., unit, integration, and acceptance) to ensure the correct functioning of the APIs and their compliance with the specified contracts.

3.6.QA and Metrics. At this step, it is crucial to determine which quality attributes (QA) and metrics should be associated with the modernized system (Trabelsi et al., 2022). According to Fávero and Affonso (2024), QA and metrics are indispensable elements of microservice-based systems, given the intrinsic nature of such systems. This investigation also revealed that code reviews, software testing, and static code analysis can assist in defect identification and correction. In parallel, metrics can be used to ascertain the efficacy of tests, the rate of defects identified, the average time to rectify defects, code coverage, and so forth. From the product's perspective, metrics can be adopted to evaluate the modernized system's reliability, performance, usability, and response time (Krause et al., 2020).

3.7.Documentation. Developers should focus their efforts on preparing documentation for the system that will be modernized in this step. In short, essential documentation artifacts must be concise and understandable, accurately reflecting the system's logical models, design, and architecture. To do so, it is recommended that these artifacts meet the following criteria: documentation coverage, target audience, and access format. The documentation must be consistently reviewed and updated throughout this step, as inconsistencies may compromise the modernization process and necessitate exclusive use of the source code for all development (Parikh et al., 2022).

Once the scope of the problem has been delineated and a preliminary solution of the modernization has been proposed, teams can proceed with the microservices **development** activity. As evidenced by the findings of the investigation conducted by Fávero and Affonso (2024), this activity can be carried out in two approaches: (i) developing new code from scratch, and (ii) extracting microservices from the legacy source code. First, the legacy code is identified as having minimal value, and developers should prioritize the preservation of the functionalities of the legacy system by implementing a new source code. Second, the legacy code is deemed valuable and can be leveraged to extract and transform the functionalities of the legacy system into equivalent microservices. Independent of the approach, it is also crucial to consider the gradual migration strategy for this activity, whereby the legacy system's previous version should be discontinued after the microservice implementation to prevent maintenance issues arising from the coexistence of functionalities in two distinct locations (Prasandy et al., 2020; Michael Ayas et al., 2021; Bandara and Perera, 2020).

4.1.Data repository. In this step, the company must prepare the development infrastructure aligned with the principles and practices of DevOps, enabling the independent and parallel development of microservices by the development teams. Such structure facilitates continuous software delivery, thereby ensuring the realization of the benefits associated with microservices-based development from both an organizational and customer perspective (i.e., those interested in modernization) (Mazzara et al., 2021).

4.2.Test cases. In this step, the execution scenarios must be delineated to evaluate whether the system or its functionalities meet the established requirements. Planning tests have been a feasible alternative in modernizing legacy systems, as it involves identifying the input conditions, the steps to be followed, and the expected results for each functionality. Therefore, it can be said that test cases serve as a basis for verifying whether the software meets the requirements and behaves as expected (Prasandy et al., 2020).

4.3.Microservice development. In this step, the microservices are developed following the design and technological specifications, as well as the business and technical requirements established in the previous steps (i.e., 2.1, 3.1 to 3.7). As a result, the objective is to integrate the microservices in a way that ensures their collective functionality. During implementation, it is recommended that developers adhere

to best practices in software engineering, such as writing clean and modular code, to ensure the scalability and resilience of the microservices. Moreover, it is also suggested that tools be used to develop, manage versions of, and deploy the microservices. Once the microservices have been developed, they are subjected to a series of tests, debugging procedures, and deployment in a production environment for use by end users (Michael Ayas et al., 2021).

4.4.Database migration. In this step, the microservice databases must be created based on the understanding gathered in Steps 1.1, 2.1 to 2.4, and 3.4. The database migration is a complex task, encompassing the transfer of both the data schema and the data itself. According to Fávero and Affonso (2024), this step must be carried out in parallel with the microservice development, since there is an intrinsic relationship between the microservice functionality and its data. Moreover, another relevant aspect to consider for a successful database migration is the data migration between the old and the new schema to ensure the integrity of information between the legacy and modernized systems (Prasandy et al., 2020).

4.5.Version control. Version control is an essential practice in microservice-based development, as it enables the tracking and management of changes to both the source code and the system configuration over time. Furthermore, version control enables the rollback to previous versions in the event of complications and facilitates the deployment of new software versions in a regulated and automated manner. In this direction, it is also recommended that versioning for the microservice contract be defined, as well as version control of its source code. To do so, development teams must create individual repositories for each microservice in this step (Freire et al., 2021).

4.6.Monolith to microservices. This step transitions the system from its legacy version to a modernized one. This transition should occur gradually as the microservices develop, allowing both systems to coexist (Bandara and Perera, 2020).

Integration is an activity that aims to combine and synchronize the microservices developed to compose the modernized system. During this activity, the microservices are linked through APIs or other communication channels, facilitating the transfer of data and information essential for the system's overall functionality. This activity may also entail the integration of the modernized system with other systems or external components to address the novel system requirements. Moreover, integration may involve the configuration of continuous deployment pipelines and process automation, to facilitate the distribution of microservices in different environments, such as development, testing, and production. Upon completion of this activity, the integrated system is expected to be prepared for deployment and made accessible for utilization by end users (Parikh et al., 2022).

5.1.Service Mesh. In this step, microservices are packaged into images and deployed into containers for insertion into the operational environment. This activity also includes defining messaging, routing, load balancing, discovery services, among others. The service mesh refers to the communication and interaction between the various microservices that make up a system, as well as the monitoring, diagnostics, and maintenance mechanisms used to ensure their health and performance (Parikh et al., 2022).

5.2.CI/CD. This step encompasses the following activities: continuous integration (CI), continuous delivery (CD), and continuous deployment. CI requires submitting code changes to be merged into the primary branch. Automated build and test processes guarantee that the code in the main branch maintains production quality. Thus, teams can identify problems related to compatibility or code conflicts, mitigating the probability of introducing errors and quality issues into the system. In CD, a production-like environment automatically receives code changes passing through the CI process. To ensure continuous delivery in a microservice environment, it is important to have a robust set of automation tools, comprehensive automated testing, and a culture of collaboration and effective communication between development teams. Continuous deployment requires that code modifications have undergone the preceding two steps (i.e., CI/CD) and are prepared for automated deployment to the production environment. A CI/CD process that is to be considered robust should include the following features: (i) the creation and deployment of microservices in an independent manner; (ii) the deployment of said microservices in environments designated for development, testing, and quality assurance; (iii) the concurrent deployment of both existing and new versions; and (iv) the packaging of images in containers within the production environment (Dehghani et al., 2022; Parikh et al., 2022).

5.3.Test. This step represents the automation of unit and integration tests for microservices. Unit tests aim to verify the individual behavior of small parts of the code, usually functions or methods, isolating them from other system parts. These tests are automated to be executed quickly and repeatedly whenever changes exist in the code, ensuring that new features do not break existing ones, and facilitating the early detection of bugs. Integration tests aim to verify the interaction between different system components, such as modules, microservices, or external systems. These

tests automate the verification of the functionality and compatibility of the system as a whole, ensuring that the individual components operate correctly together. By automating unit and integration tests, development teams can proactively identify and correct problems, reducing the time needed to validate code changes and increasing the reliability and stability of the software. To automate tests, it is recommended to use specific testing tools, development frameworks, and continuous integration practices, which facilitate the automatic execution of tests in controlled environments and the generation of detailed reports on test results (Parikh et al., 2022; Prasandy et al., 2020).

Monitoring is an essential activity in microservice-based systems because of the distributed and complex nature of these systems. Therefore, monitoring in microservices environments involves collecting and analyzing performance-related metrics, such as response time, error rate, resource usage (e.g., CPU, memory, network), and availability. Such metrics are collected at runtime and can be visualized through dashboards and reports, providing valuable insight into the system's health and helping to identify bottlenecks, failures, and optimization opportunities. Furthermore, monitoring activities can include the early detection and alerting of anomalies and potential problems, thereby enabling operations and development teams to take corrective action expeditiously and minimize the impact on end users. The implementation of a monitoring system in microservice environments necessitates the utilization of specialized tools and platforms for the collection, storage, and analysis of monitoring data, as well as observability practices aimed at enhancing system comprehension and diagnosis in the event of problems (Ma et al., 2022a).

6.1.Dashboard. This step involves implementing dashboards to provide centralized, continuous feedback on metrics and microservice's health (Freire et al., 2021). To do so, a set of tools can be utilized as dashboards and, simultaneously, facilitate the observability of the modernized system, collecting a set of metrics, trace information, and log (Tozzi, 2022). Metrics can be defined as a logical meter used to measure and record data over a specified period. Trace information refers to data associated with the life cycle of a single transactional object in a system, which can be transaction specific or related to other aspects of the system's operation. Logs deal with discrete events that occur while running a system, such as error messages, audit events, or request-specific metadata. Among the tools referenced in the literature, Grafana[2] has distinguished itself as a valuable resource and an open-source solution.

6.2.Issue alerts. This step addresses the processing of warning messages (or alerts) when the system transitions to a critical or attention state. The alerts may assist in the early detection and mitigation of problems to prevent microservice disruption. Alerting represents a key practice to microservice monitoring, as it enables immediate notification to operations and development teams of significant events or issues that could impact system availability or performance (Freire et al., 2021).

6.3.Maintenance. This step requires the collaboration of development teams to ensure the coordination and maintenance of preventive, corrective, and improvement actions based on the data collected from monitoring (i.e., dashboard). A development team needs to be aware of the following maintenance activities: maintaining the latest software versions for microservices, monitoring and fixing bugs that affect microservice behavior or functionality, identifying and resolving performance issues in microservices, assessing and adjusting microservice capacity to handle traffic spikes and increased demand, and finally, maintaining constant monitoring to proactively identify problems and anomalies (Colanzi et al., 2021b).

6.4.Rollback. This step suggests that development teams can rollback microservices (or system versions) in case of failure. This enables the system to revert to the legacy and stable environment, thus facilitating the performance of effective maintenance. Therefore, it can be said that enabling rollback is an important practice during the microservice life cycle because it provides a way to undo changes that have caused problems or introduced regressions in the system. By incorporating rollback strategies as a maintenance approach, teams can reduce the risk of system failures and outages, and ensure a more reliable and consistent user experience (Colanzi et al., 2021b).

4 CASE STUDY

This section presents a case study conducted to evaluate the applicability, strengths, and weaknesses of Micro4Delphi. The subject of the empirical analysis is a legacy system developed in Delphi that addresses scholar evaluation, referenced from this point on forward as **Avance**. Next, an overview of the subject application and the empirical strategies adopted for conducting this case study will be provided.

Subject Application. The **Avance** system was developed in Delphi 10.4 with a Firebird database version 2.5. The system was organized in layers and modules, with distinct layers separating the user in-

[2]https://grafana.com

terface components (i.e., forms), the business logic of the system (i.e., units), and the database connection (i.e., data module). In terms of size, this system has 15 forms, six reports, 11 data modules, 10 database tables, and 19 components of connectivity to the database, including data sources, tables, and queries (see Figure 2 – **Side A**).

In terms of functionality, the **Avance** system enables users (i.e. students and teachers) to register with personal data so that they can authenticate and perform operations according to each profile. The system was designed to distinguish between different classes and the students enrolled in each one. In the teacher's profile, the system enables each one to prepare lessons for each class based on the content that must be registered in the system. Moreover, teachers can also prepare exercises for each lesson, which must be completed by the students during the class period. Regarding the student's profile, the **Avance** system enables each student to monitor the content they have learned and the exercises they have completed, aiming to facilitate the learning process. To do so, the system enables the viewing of the number of correct and incorrect answers in each exercise, as well as the recommended solution for each exercise.

Empirical Strategy. Figure 2 shows the modernization of the **Avance** system to MSA, which was based on the analysis of the system source code and its database. The illustration on the left side (A) shows the organization of the **Avance** system in three layers: user interface, business logic, and data access layer. The illustration on the right side (B) shows the microservices that emerged from the modernization process, namely: `MSClass`, `MsSchool`, `MsLesson`, and `MsUser`. Moreover, the illustration on side (B) presents the organizational structure of the `MsUser` microservice and the MSA elements in the bottom part (i.e., `Log`, `Config`, and `Catalog` services).

In order to demonstrate the behavior of Micro4Delphi, only the tasks related to the transition from the legacy system to the `MsUser` microservice will be presented in this section due to space limitations. Therefore, it can be stated that after the **planning activity** (Step 1.1), the system was comprehended and a layered model was designed, as illustrated on the left side (A) of Figure 2. The remaining steps of this activity were not carried out because of the limitations in the size of the team responsible for conducting this case study.

In the **analysis activity**, each of the prescribed steps was visited. Initially, the analysis involved the construction of an understanding regarding the legacy system and its database, besides the business model associated with such a system (Steps 2.1 to 2.3). In this sense, some user-related functionalities were identified, including, for instance, user authentication. Additionally, database tables, such as `User_Access` and `User`, were identified as potential sources of support for the `MsUser` microservice. Since the purpose is to transform the legacy system into microservices, it was decided that the user interface should be maintained and the native language of the system should be preserved (Steps 2.3 and 2.4).

In the **decomposition activity**, the legacy code must be analyzed to identify candidates for microservices (Step 3.1). As illustrated in Figure 2, **Side A**, four microservices candidates were identified: `MsClass`, `MsSchool`, `MsLesson`, and `MsUser`. Since the **Avance** source code has proven to be of high value, it is recommended that the system be refactored so that its source code and data schema can be reused in the microservices development stage (Steps 3.2 to 3.4). Next, the equivalent APIs are modeled for the identified microservices, which focus on defining how these microservices will communicate with one another and/or components external to the system (Step 3.5). Finally, the documentation of these microservices via Swagger must be completed (Step 3.7), as it aims to provide interested parties with information on how to use them.

During the development phase, microservices were implemented, as illustrated in Figure 2. As illustrated, each microservice incorporates its own database and necessitates a dedicated database connection unit, as detailed in Listing 1. With microservice decoupling from the user interface, the database connection components were implemented on the command line. In Line 3, the name of the database connection is defined, which is then used by the `cnxDef` connection component instantiated in Line 6. Between Lines 10 and 12, the database connection parameters (i.e., `user`, `password`, and `database`) are defined for each microservice. Finally, it is important to note that the comments in Lines 2, 4, 8, 13, and 16 represent omitted code sections that do not require configuration details.

From this point forward, only an overview of the organization of the `MsUser` microservice is provided because of the space limitations of this paper. The `model` package contains a class designated User (`Model.User.pas`). The `dao` package contains the `DAO.User.pas` class, the purpose of which is to deal with data persistence. To do so, this class implements the `getUser` method to query users and the `postUser` method to include users in the database. The `util` package contains a set of utility classes for the `MsUser` microservice to function. A comprehensive examination of the classes in the aforementioned packages is

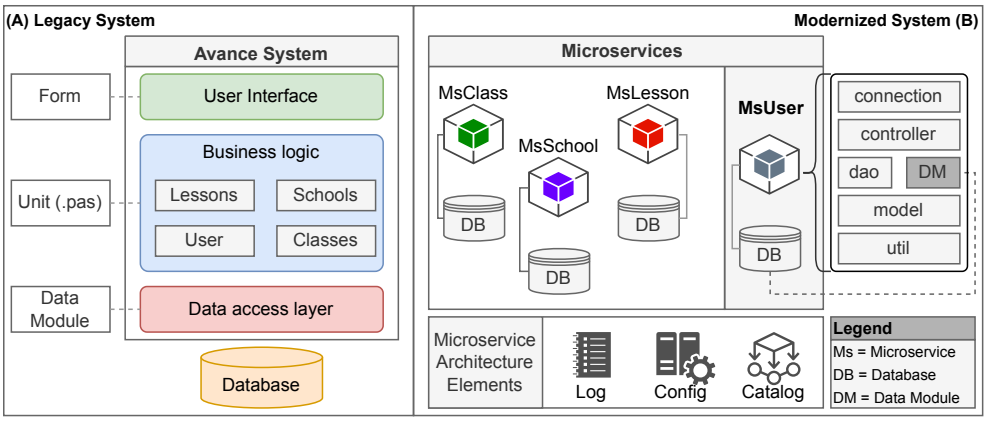

Figure 2: Legacy System Modernization for MSA.

Listing 1: Connection unit.

```
1  unit udmUser;
2  //...
3  const CON_NAME = 'User_runtime';
4  //...
5  procedure TdmFiredac.FDManagerBS(Sender:
       ↪ TObject);
6  var cnxDef: IFDStanConnectionDef;
7  begin
8    //...
9    cnxDef.Name := CON_NAME;
10   cnxDef.Params.UserName := 'sysdba';
11   cnxDef.Params.Password := 'masterkey';
12   cnxDef.Params.Database :=
         ↪ 'C:\Avance\User.fdb';
13   //...
14   cnxDef.Apply;
15 end;
16 end.
```

outside the purpose of this paper.

The `controller` package contains the `Controller.User.pas` class in the code fragment shown in Listing 2. In short, the purpose of this class is to facilitate interactions between the user interface and the `dao` layer for data persistence. To illustrate such interactions, this controller implements two procedures as GET (Line 17) and POST (Line 26) methods. In both methods, it is possible to observe the creation of the `DAOUser` object from the `dao` layer in Lines 20 and 30, respectively. In Lines 22 and 32, the calls to the `getUser` and `setUser` methods of `DAOUser` are presented, both of which have been previously described (`dao` package).

It should be highlighted that the `MsUser` microservice development activity was concluded with the documentation of the APIs via Swagger between Lines 7 and 10 (Listing 2). Specifically, Line 7 delineates the method that has been documented (i.e.,

Listing 2: Controller User.

```
1  unit Controller.User;
2  //...
3  type
4    //...
5    public
6      //...
7      [SwagPOST('', 'Post', true)]
8      [SwagResponse(200, TModelUser,
           ↪ 'Success')]
9      [SwagResponse(400, TModelResponse,
           ↪ 'Bad Request')]
10     procedure post;
11     //...
12     constructor Create(Req: THorseRequest;
           ↪ Res: THorseResponse);
13   end;
14
15 implementation
16
17 procedure TControllerUser.get;
18 //..
19 begin
20   DAOUser := TDAOUser.Create;
21   //...
22   FResponse.Status(200).Send<TJSONArray>
         ↪ (DAOUser.getUser);
23   //...
24 end;
25
26 procedure TControllerUser.post;
27 //...
28 begin
29   User := getBody;
30   DAOUser := TDAOUser.Create;
31   //...
32   FResponse.Status(200).Send<TJSONObject>
         ↪ (DAOUser.setUser(User));
33   //...
34 end;
35 end.
```

`POST`), with the response messages associated with the respective codes (e.g., `200` and 'Success') detailed in Lines 8 and 9. The documentation for the `get` method follows a similar structure; however, it has been omitted because of space limitations.

Therefore, it can be stated that, upon conclusion of the **development activity**, Steps 4.1 to 4.5 were executed in full in this case study, as the microservice is now prepared for deployment in its execution environment. Concerning the other activities of the Micro4Delphi process, it should be noted that the **integration and monitoring activities** were partially conducted. Of particular note are the development of tests (Step 5.3) through the Swagger API and the monitoring of microservices in the execution environment (Step 6.1). Although these activities are relevant to the modernization process, they do not address the transition from legacy systems to microservices and, therefore, will not be detailed in this section.

5 CONCLUSIONS

This paper presented the Micro4Delphi, a process designed to support the modernization of legacy systems developed in Delphi to MSA. To do so, each activity has a set of steps targeted to the legacy system and organization of development teams or company structure. Given the scenario described, the principal contributions of this paper are outlined as follows.

The process proposed in this paper may prove advantageous for the development and software engineering areas. To the best of our knowledge, it represents the first initiative that can effectively support the modernization of legacy systems to MSA (Fávero and Affonso, 2024). This process has the potential to serve as a theoretical framework that can contribute to the formulation of new initiatives or improvements to existing ones.

The case study presented in Section 4 provides an overview of the modernization of legacy systems in Delphi, emphasizing the organization of the modernized system based on microservice organization. In this sense, it is notable that this case study can serve as a reference for others interested in transforming their systems for the current computing scenario.

Regarding future work on Micro4Delphi, at least three activities are planned: (i) the conduction of additional case studies, which will enable a more comprehensive evaluation of this process across a range of software domains, including web, mobile, and enterprise; (ii) the instantiation of Micro4Delphi for other programming languages, which will facilitate an evaluation of its behavior when a new development environment is used; and (iii) the use of this process in an industrial setting, which will allow for an evaluation of its behavior when applied in a larger, real-world development and execution environment.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES).

REFERENCES

Almeida, N. R., Campos, G. N., Moraes, F. R., and Affonso, F. J. (2024). Modernization of legacy systems to microservice architecture: A tertiary study. In *The 23rd International Conference on Enterprise Information Systems.*, pages 1–12. INSTICC, SciTePress.

Auer, F., Lenarduzzi, V., Felderer, M., and Taibi, D. (2021). From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137.

Bandara, C. and Perera, I. (2020). Transforming monolithic systems to microservices - an analysis toolkit for legacy code evaluation. In *20th International Conference on Advances in ICT for Emerging Regions, ICTer 2020 - Proceedings*, page 95 – 100. Institute of Electrical and Electronics Engineers Inc.

Colanzi, T., Amaral, A., Assunção, W., Zavadski, A., Tanno, D., Garcia, A., and Lucena, C. (2021a). Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices. In *The 15th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 61–70, New York, NY, USA. Association for Computing Machinery.

Colanzi, T., Amaral, A., Assunção, W., Zavadski, A., Tanno, D., Garcia, A., and Lucena, C. (2021b). Are we speaking the industry language? the practice and literature of modernizing legacy systems with microservices. In *ACM International Conference Proceeding Series*, page 61 – 70. Association for Computing Machinery.

Dehghani, M., Kolahdouz-Rahimi, S., Tisi, M., and Tamzalit, D. (2022). Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. *Software and Systems Modeling*, 21(3):1115 – 1133.

Dragoni, Nicola, Giallorenzo, S., and Lafuente, A. L. (2017). Present and ulterior software engineering. In *Microservices: Yesterday, Today, and Tomorrow.*, pages 195–216. Springer.

Embarcadero (2024). Native apps for any device from one codebase with delphi! on-line. Avaliable in: https://www.embarcadero.com/products/delphi, accessed on March 20, 2025.

Erl, T., Balasubramanian, R., Pautasso, C., Wilhelmsen, H., Carlyle, B., and Booth, D. R. (2012). *SOA with REST*. Prentice Hall, Philadelphia, PA.

Freire, A. F. A. A., Sampaio, A. F., Carvalho, L. H. L., Medeiros, O., and Mendonça, N. C. (2021). Migrating production monolithic systems to microservices using aspect oriented programming. *Software - Practice and Experience*, 51(6):1280 – 1307.

Fávero, L. F. and Affonso, F. J. (2024). A systematic mapping study of legacy system modernization to msa. on-line. https://drive.google.com/file/d/1-sKo1mU54Q6QQY1l7slfuyxx0PwtxdmA/view?usp=sharing, acessed on March 20, 2025.

Krause, A., Zirkelbach, C., Hasselbring, W., Lenga, S., and Kroger, D. (2020). Microservice decomposition via static and dynamic analysis of the monolith. In *The IEEE International Conference on Software Architecture Companion, ICSA-C 2020*, page 9 – 16. Institute of Electrical and Electronics Engineers Inc.

Kyryk, M., Tymchenko, O., Pleskanka, N., and Pleskanka, M. (2022). Methods and process of service migration from monolithic architecture to microservices. In *16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, TCSET 2022*, page 553 – 558. Institute of Electrical and Electronics Engineers Inc.

Lewis, J. and Fowler, M. (2019). Microservices guide. on-line. Avaliable in: https://martinfowler.com/microservices, accessed on March 20, 2025.

Li, C.-Y., Ma, S.-P., and Lu, T.-W. (2020). Microservice migration using strangler fig pattern: A case study on the green button system. In *Proceedings - 2020 International Computer Symposium, ICS 2020*, page 519 – 524. Institute of Electrical and Electronics Engineers Inc.

Ma, S.-P., Li, C.-Y., Lee, W.-T., and Lee, S.-J. (2022a). Microservice migration using strangler fig pattern and domain-driven design. *Journal of Information Science and Engineering*, 38(6):1285 – 1303.

Ma, S.-P., Lu, T.-W., and Li, C.-C. (2022b). Migrating monoliths to microservices based on the analysis of database access requests. In *IEEE International Conference on Service-Oriented System Engineering, SOSE 2022*, page 11 – 18. Institute of Electrical and Electronics Engineers Inc.

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T., and Dustdar, S. (2021). Microservices: Migration of a mission critical system. *IEEE Transactions on Services Computing*, 14(5):1464 – 1477.

Michael Ayas, H., Leitner, P., and Hebig, R. (2021). The migration journey towards microservices. *Lecture Notes in Computer Science*, 13126 LNCS:20 – 35.

Newman, S. (2021). *Building microservices*. O'Reilly Media, Sebastopol, CA, 2 edition.

Osman, M. H., Saadbouh, C., Sharif, K. Y., Admodisastro, N., and Basri, M. H. (2022). From monolith to microservices: A semi-automated approach for legacy to modern architecture transition using static analysis. *International Journal of Advanced Computer Science and Applications*, 13(10):907 – 916.

Parikh, A., Kumar, P., Gandhi, P., and Sisodia, J. (2022). Monolithic to microservices architecture - a framework for design and implementation. In *International Conference on Computer, Power and Communications, ICCPC 2022 - Proceedings*, page 90 – 96. Institute of Electrical and Electronics Engineers Inc.

Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18.

Prasandy, T., Titan, Murad, D. F., and Darwis, T. (2020). Migrating application from monolith to microservices. In *Proceedings of 2020 International Conference on Information Management and Technology, ICIMTech 2020*, page 726 – 731. Institute of Electrical and Electronics Engineers Inc.

Pressman, R. and Maxim, B. (2019). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education. 9th Edition.

Richardson, C. (2018). *Microservices patterns*. Manning Publications Company.

Sellami, K., Ouni, A., Saied, M. A., Bouktif, S., and Mkaouer, M. W. (2022). Improving microservices extraction using evolutionary search. *Information and Software Technology*, 151.

Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232.

TIOBE (2024). Tiobe index for november 2024. on-line. Avaliable in: https://www.tiobe.com/tiobe-index, accessed on March 20, 2025.

Tozzi, C. (2022). The 3 pillars of observability: Logs, metrics and traces. Available: https://www.techtarget.com/searchitoperations/tip/The-3-pillars-of-observability-Logs-metrics-and-traces, Accessed on March 20, 2025.

Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S., and Guéhéneuc, Y.-G. (2022). From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*.