Service Extraction from Object-Oriented Monolithic Systems: Supporting Incremental Migration

Soufyane Labsari, Imen Sayar, Nicolas Anquetil, Benoit Verhaeghe, Anne Etien

Service Extraction from Object-Oriented Monolithic Systems: Supporting Incremental Migration

Soufyane Labsari*, Imen Sayar*, Nicolas Anquetil*, Benoit Verhaeghe†, Anne Etien*

*Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, Lille, France
†Berger-Levrault, Limonest, France

***Abstract*—Migrating large monolithic systems to service-based architecture is a complex process, mainly due to the difficulty of extracting reusable functionality from tightly coupled components. To support this, Service Identification techniques have been proposed to decompose monoliths into service candidates. Implementing these service candidates requires significant restructuring efforts. To address this complexity and build confidence in the target architecture, prior research recommends using an incremental migration approach where services are extracted one at a time. However, incremental migration has been poorly explored in the literature and lacks dedicated tool support. Thus, we explore the idea of a tool-assisted service extraction to support incremental migration, where one service is extracted at each increment. This paper first discusses the challenges associated with incremental migration. Then, it presents a model-based extraction approach aimed at automatically extracting functionality as a service. The approach is supported by a tool prototype evaluated on an industrial system and an open-source project. Our results show that our tool can extract standalone services that are successfully invoked by a reduced version of the monolith.**

***Index Terms*—Monolithic System, Incremental Migration, Service Identification, Service-based Architecture, Strangler Fig**

I. Introduction

Monolithic systems embed all their functionalities in a single unit, and as they grow larger, they can become increasingly difficult to manage [1]. In that context, organizations migrate their monolithic systems to service-based architectures mainly to reduce maintenance costs and improve flexibility [2, 3].

Monolith migration to services-based architecture relies on analyzing and reusing the valuable business logic contained in the monolith [4]. To support this, many Service Identification (SI) techniques have been proposed to identify reusable functionalities that can be transformed into services [5, 6]. These techniques use various monolith assets such as source code, developer expertise, documentation, and existing tests to analyze and recommend distinct functionalities that can be extracted and deployed as standalone services.

Implementing a service-based application from a set of service candidates derived from a large monolithic system is a complex and time-consuming process [3]. It requires significant re-architecture to isolate functionalities, decouple dependencies, and restructure the system into standalone, and deployable services. The lack of developer expertise in service-based architecture [2] and the need to maintain business continuity during migration [7] add further complexity to this task.

To address this complexity, build confidence, and minimize business disruption, previous work recommends to use an incremental migration approach where services are implemented one at a time [8, 9, 10]. However, they do not go further than recommendations and do not present challenges or describe approaches. Only Li et al. [11] present a manual approach to incrementally extract services and apply it to an open-source project. While the proposed process is outlined, no tool or automation support is provided.

Moreover, several surveys among industrial practitioners [2, 12, 13] show that developers involved in migration projects tend to use an incremental approach to migrate their system to a service-based architecture. The use of incremental migration approaches by industry professionals, combined with the lack of research focus on these approaches, creates a gap.

To bridge this gap between the research and industrial worlds, we present our preliminary work on an incremental migration approach based on service extraction. Our goal is to provide a tool-based approach to gradually migrate from monolithic to service-based architecture. We propose an approach, where, in each increment, a service is extracted from the monolith and coexists alongside a reduced version of the monolith (i.e., with removal of extracted code). Such an approach involves assisting developers during the different steps of the migration as described by Newman in [14]; namely service identification, service extraction, call redirection, and removal of old code in the monolith.

To achieve this goal, we developed a semi-automated service extraction approach based on a model representation of the monolith. The approach computes a static call graph from the functionality entry point specified by the user, collects and filters relevant classes, performs recursive dependency analysis, and generates the code for the extracted service.

The main contributions of this paper are the following:

- a discussion of the challenges associated with incremental migration towards services;
- an approach that focuses on identifying and extracting one service at a time to support incremental migration;
- a five staged process for identifying one service;
- a prototype implementing our approach.

The rest of the paper is structured as follows. Section II presents the related work and describes the used incremental

migration model. Section III discusses the challenges associated with incremental migration. Section IV describes our proposed service extraction approach. The application of our extraction approach on an industrial case and an open-source project is presented in Section V. A discussion about our work is given in Section VI. Finally, Section VII concludes the paper and discusses future works.

II. Related Work and Background

In this section, we discuss the related work on service extraction and incremental migration. We then present the incremental migration model on which we rely. In the remainder of this section, we use the term *legacy* [7] as it is used in the related work to refer to the monolith.

A. Related Work on Service Extraction

Several methods have been proposed to reuse object-oriented legacy code by extracting it as services. This is closely related to the topic of Service Identification [6].

Bao et al. proposed an approach for extracting services from object-oriented legacy systems using UML use cases to identify services [15]. Their hypothesis is that a UML use case is a good service candidate. The approach begins with manually writing UML use cases. In the next step, test cases are generated from each use case. After an instrumentation step, tests are run to produce execution traces. From a manual analysis of these traces, code segments implementing the use case are collected. Zhang et al. suggested extracting services using formal concept analysis, a technique for identifying relationships between software artifacts and their properties combining it with concept slicing [16]. Although both approaches enable the extraction of a single service, several of their steps are manually performed, which can be challenging for large systems.

According to Abdellatif [6], *the state-of-the art of the Service Identification Approaches (SIAs) are still at their infancy mainly due to (1) the lack of validation on real enterprise-scale systems; (2) the lack of tool support, and (3) the lack of automation of SIAs*.

B. Related Work on Incremental Migration

Incremental migration is a process that consists of a sequence of increments, each representing a distinct, and manageable transformation step toward a new system [17]. This allows system operations to continue during migration, reduces risk, and makes complexity more manageable by migrating in smaller, more controllable steps [7]. Incremental migration is frequently used in the industry [2, 12, 13]. It is also a recommended approach for managing the complexity of migrating large monolithic systems to service-based architectures [8, 9].

A first incremental migration approach called *Chicken Little* was proposed in the '90s by Brodie et al. [17]. This approach is similar to the Strangler Fig [18] defined 11 years later. Li et al. [11] described an incremental migration process based on the Strangler Fig model [18]. Microservices candidates are identified using Domain-Driven Design (DDD) once for the whole system. Then metrics are used to define the order in which services should be extracted. The approach is applied to a small case study. It is not clear how the whole approach scales. This approach decomposes the whole system once and uses this decomposition to extract services during the entire migration process, which can last several years [17]. This can lead to a lack of flexibility during the migration as the initial decomposition could become obsolete. More generally there is no evidence about the practicability of using a one-shot decomposition to drive an incremental migration. Most of the SIAs focus on decomposing the monolith in one shot.

As the Strangler Fig model is reported to be the most widely used incremental approach in the industry [2, 12], we will focus on this approach in the remainder of this paper.

C. Background

In the context of legacy system modernization, Martin Fowler proposed the Strangler Fig model. This incremental approach suggests that the new system should be incrementally built around the old, eventually "strangling" and replacing the old system components as the new system (the strangler application) takes over. In the context of migration to service-based architecture, the approach begins by identifying functionality that can be extracted as a service and migrated to the strangler application, gradually replacing the monolith with new services (see Figure 1). As the process continues, the monolith shrinks while the new system becomes more robust [19].

More specifically, each increment of the migration is implemented sequentially following a four-step process; three main steps [14] and an optional one [19]:

1) Service identification: A functionality is selected, and the corresponding code must be identified within the monolith.

2) Service extraction: The functionality is extracted (copy or reimplement the code) to the service architecture and may require adaptation to work as an autonomous service.

3) Calls redirection: Calls to the functionality are redirected to the extracted service.

4) Manage residual code (optional): Once the service is deployed and calls are redirected to this service, the extracted code that remains in the monolith can be deleted if the monolith is still being maintained during the migration.

The challenges associated with these steps are not discussed. In addition, no tool supports this approach.

III. Incremental Migration Challenges

This section describes the challenges we have identified for each step (see Section II-C) of the Strangler Fig migration model.

A. Service Identification Step

Challenge C1: Service Prioritization [11]. Selecting the functionality to extract requires special care. Removing a highly dependent service too early could complicate the next steps and may require additional increments to be completed before the current increment can be deployed.

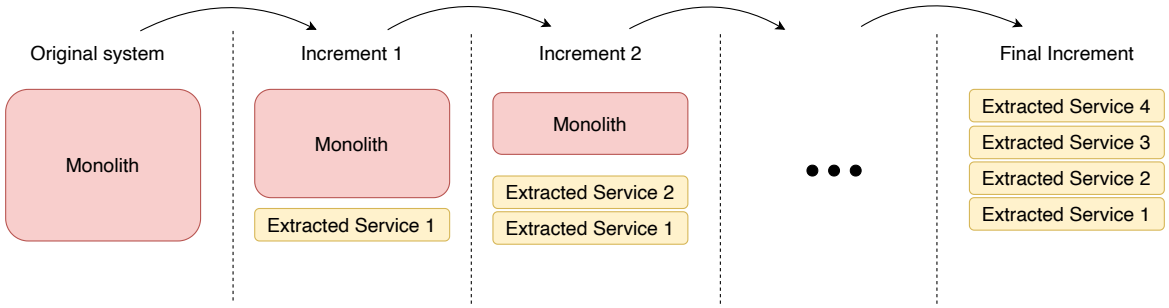

Fig. 1. Strangler Fig in the context of migration to service-based architecture

Challenge C2: Service Granularity [20]. Determining the appropriate service granularity is a key challenge. Service granularity refers to how many functionalities each service should encapsulate. Finding the right balance is critical to system efficiency. On the one hand, if services are too coarse-grained, they may contain too many responsibilities, resulting in reduced modularity and limiting the benefits of independent development and scaling. On the other hand, services that are too fine-grained can lead to excessive inter-service communication, resulting in increased network latency and increased complexity in managing dependencies.

Challenge C3: Functionality code identification. In Object-Oriented (OO) programs, the code implementing a functionality is distributed across many interrelated objects [15], making its identification challenging. Moreover, the dynamic features of object-oriented languages, such as *reflection* [21], introduce runtime dependencies that are difficult to analyze [22].

Challenge C4: Service interdependencies. The high degree of coupling between components of the monolith makes it difficult to understand dependencies between functionalities and challenging to identify clear service boundaries [12].

B. Service Extraction Step

Challenge C5: Shared code. The extracted functionality might still rely on other parts of the monolith, such as shared classes (e.g., utility classes, models) or utility methods. Copying these objects could spread duplicated code among both the extracted service and the monolith, and further complicate maintenance efforts.

Challenge C6: Shared state. The functionality may depend on shared states in the monolith. Shared states between functionalities within a monolith refer to data or variables that multiple parts of the system access and modify, creating dependencies and coupling between those functionalities [19]. Managing state transitions from monolith to service and ensuring data consistency can be a significant challenge [23].

Challenge C7: Service adaptation. The extracted service should be adapted to the service-oriented technology stack that might be different. The functionality must be refactored to use this technology stack, which may involve changes to frameworks, programming languages, and data formats.

C. Calls Redirection Step

Challenge C8: Calls redirection. Redirecting calls involve multiple scenarios: (1) from the monolith to the newly extracted service, (2) from the extracted service back to the monolith, or (3) from the extracted service to previously extracted services. Each of these scenarios requires rigorous tracking and understanding of the dependencies within both the monolith and the previously extracted services [14, 19].

D. Manage Residual Code Step

Challenge C9: Dead code removal. The removal of legacy code, as new services are extracted, requires careful coordination to avoid introducing bugs or leaving dead code in the monolith, which can lead to maintenance issues. If the code implementing the functionality is no longer referenced elsewhere in the monolith, this code can be marked as dead and safely removed.

E. General Challenge

Challenge C10: Dealing with several increments. Extracting a service after several others leads to the challenges previously cited (**C2**, **C5**, **C6**, and **C8**), not only with the monolith but also with each single previously extracted service. Service extractions are local and may contain code artifacts that are shared with other services. This could result in a wrong service cut, making maintenance of the services even more challenging. Brodie et al., highlight the dependencies between increments and that independent increments can be implemented in any order [17].

Addressing these challenges by providing tool support is crucial for the success of incremental migration, as it ensures a smoother transition from monolithic to service-based architecture without degrading system integrity.

IV. Proposed Approach to Extract Service

This section presents our approach to support incremental migration and explains how we start dealing with some of the identified challenges (see Section III). Our approach aims at extracting all the functionality's relevant artifacts within an OO monolith using static analysis. In its current version, our approach focuses on challenges **C3** and **C4** and consists of five stages (see Figure 2): (1) select the functionality to extract, (2)

compute the associated call graph, (3) collect classes, (4) filter classes, and (5) collect dependencies.

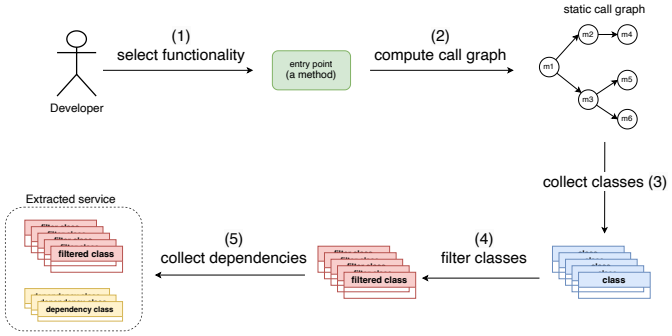

Fig. 2. Overview of our functionality extraction process

Stage 1: Select functionality to extract

In the context of an increment, one must select the functionality to extract. In our solution, we propose to let the developers choose the functionality that will be extracted as a service. To help them in this task, we only require that they select the entry point method (or methods) of the functionality.

Stage 2: Compute call Graph

From the given entry point, one must identify all the code implementing the functionality (Challenge **C3**). For this, we compute the call graph of the entry point to catch all the methods involved in the implementation of the functionality. This is done using a static approach that ensures soundness [24] (i.e., catch all the necessary methods for any scenario of the functionality). However, static approaches are known to lack precision (i.e., unnecessary methods may also be identified as false positives) [25]. As a way to improve precision, we use Variable Type Analysis (VTA) [26]. This approach supports context-sensitive call graph construction, i.e., it differentiates between the different contexts in which a method is invoked, improving accuracy, especially in cases involving polymorphic calls.

Stage 3: Collect classes

In this stage, we collect the classes containing the methods present in the previously computed call graph. These methods are selected within their classes in preparation for the next stage.

Stage 4: Filter classes

In the classes collected at the previous stage, not all class members (attributes and methods) are relevant to the functionality to extract. To improve the precision of the extraction, we proceed by eliminating the irrelevant members. This will also ensure that we remove unnecessary dependencies (Challenge **C4**). Class attributes used by the selected methods are also selected as part of the functionality implementation. This ensures that all relevant data and states associated with the functionality are captured. Possible setters and getters for these attributes are also selected, even if they do not appear in the call graph. This is needed because they might be necessary for external frameworks (e.g., Spring or Hibernate).

Stage 5: Collect dependencies

A dependency analysis is then conducted to gather classes referenced by the filtered classes and their selected methods. For example, a method could reference a class using the Java operator `instanceOf`. Such a class would not have been caught in the previous stages.

Our approach is automated and implemented[1] in Moose[2], a data and software analysis platform. Moose is used to create a model representation of the monolith, enabling the automated analysis and extraction of specific functionalities.

V. Case Study

To assess the feasibility of our extraction approach, we conducted case studies on two Java client-server applications. The first case study focuses on the extraction of a service on a real industrial application. The second case study focuses on extracting multiple services for the migration on an open-source project. These case studies focus on migrating the server side of the applications. The server side of both applications follows a three-tier architecture composed of a presentation layer, a business layer, and a data layer. Business functionalities are exposed to the client side through the presentation layer. We use these exposed functionalities as entry points to extract services.

A. Case Study 1: Omaje

Berger-Levrault is a software publisher that provides software solutions for public administration, healthcare, education, and industry. Omaje is a subscription management Java application used internally by the company to manage the distribution of software licenses. The server side is composed of 427 classes, including 3956 methods with a total of 34.4 K lines of code.

A model representation of Omaje's source code was created in Moose, and the following work is carried out on this model. The goal of this case study is to assess the feasibility of our extraction approach (see Section IV).

Extraction of a Service: **Stage 1:** A functionality was selected for extraction as the first increment of the migration, with the associated entry point in the presentation layer being a method named `findProduct`. This functionality is responsible for fetching a product from the database given an ID (an integer). **Stage 2:** From the method entry point of the functionality, a call graph was computed and contained 49 unique methods. **Stage 3:** A total of 17 classes, 5 interfaces, and 1 enumeration were collected from the call graph methods. **Stage 4:** During the filtering step, 342 methods and 114 attributes were removed from the 23 types collected in stage 3. 84 methods and 59 attributes were kept.

[1]https://doi.org/10.5281/zenodo.14057370
[2]https://github.com/moosetechnology

The code of the extracted service was generated as a Spring project. It was successfully compiled without modification. Three test cases were manually written to cover different scenarios of the extracted functionality, and all of them passed successfully.

B. Case Study 2: Spring PetClinic

Spring PetClinic[3] is an open-source Java Web application designed to demonstrate the capabilities of the Spring Framework. Spring PetClinic simulates a veterinary clinic that allows users to manage information about pets, owners, and visits, making it a representative example of traditional monolithic architecture. The Spring PetClinic server-side is composed of 26 classes with a total of 111 methods (683 lines of code). The presentation layer exposes 16 endpoints (methods annotated with `@GetMapping` or `@PostMapping`), which are accessible to the client side through HTTP requests.

The goal of this case study is to assess the feasibility of the Strangler Fig model using our extraction approach. Several services were extracted using the Strangler Fig steps described in Section II-C. We used our extraction approach to automate the first step (*service identification*) and conducted the others manually since it is an in progress approach. The results of this case study are available in the replication package[4].

TABLE I
MIGRATION INCREMENTS OF SPRING PETCLINIC

		Extracted Service		Monolith	
Increment	**Entry point method**	**NOC**	**NOM**	**NOC**	**NOM**
0	-	-	-	26	111
1	`showOwner`	10	21	26	111
2	`showVetList`	11	21	26	108
3	`processFindForm`	12	41	26	105

Table I presents the increments performed and the associated endpoint methods extracted (second column). It shows the evolution of the monolith as services are extracted, including the number of classes (NOC) and the number of methods (NOM). Note that because the extracted classes are filtered, they may not contain all the methods and attributes they previously contained in the monolith.

First increment: **Step 1.** From the `showOwner` entry point method, 8 classes were identified including 6 classes representing data models, 1 class from the presentation layer, and 1 class from the data access layer. **Step 2.** These classes were extracted in a service project, and 2 classes were added to make this service run as a standalone Spring application, resulting in a total of 10 NOC. **Step 3.** A method was added in the monolith to perform the call redirection to the extracted service. **Step 4.** Only one method was removed from the monolith as other methods are still used.

From increment 0 to increment 1, the number of methods in the monolith did not decrease because a method was added to redirect the call to the service. The next two increments were done in the same way. Extracted services mainly contain data models that are duplicated among the monolith and the services. After the third increment, 3 services are deployed in Docker containers and successfully invoked by the monolith.

VI. DISCUSSION

Our work yielded the following conclusions: (1) many challenges are not specified explicitly in the state-of-the-art, (2) existing works on service extraction from monoliths focus mainly on decomposing the whole system, and (3) even though the incremental migration approaches are used in industry, their automation is not supported. Our current work addresses these three points and the obtained first results show that our prototype is able to extract services gradually and needs to be improved.

A notable observation from the case studies is the duplication of data models in the extracted services. This highlights the challenge of shared code (**C5**) between the monolith and extracted services. This could be handled by creating a shared library as suggested by Newman [14], meaning that a preprocessing step could be required on the monolith to prepare the incremental migration.

When extracting a service with our methodology, it is hard to identify dependencies between different monolith components, which adds difficulties to the developer in managing code duplication, calls redirection, etc. We think about using existing tools such as visualization ones to assist the developer during the extraction process.

VII. CONCLUSION AND FUTURE WORK

In this paper, we proposed an approach to support incremental migration of object-oriented monolithic systems to service-based architecture. We first discussed the challenges that developers may face when adopting the Strangler Fig model. We then presented a preliminary method to extract a functionality based on code analysis to assist developers in the implementation of an increment in the migration process. We have implemented an open-source prototype of our tool in the Moose platform. We evaluated our tool-based approach by conducting case studies on two Java projects, an open-source project (JPetStore) and a real-world project (OMAJE) from our industry partner. The case studies have shown that our prototype is able to extract standalone services that can be executed and invoked from the monolith.

Future work. In the future, we plan to use existing solutions to deal with some of the described challenges (**C1**, **C4–9**). For instance, the challenge **C4** could be addressed using formal concept analysis [16] to understand dependencies among the monolith. We will implement these solutions into a tool to better support incremental migration. We will perform a validation of our tool on real enterprise-scale systems with our industry partner.

[3]https://github.com/spring-projects/spring-petclinic
[4]https://doi.org/10.5281/zenodo.14057370

References

[1] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[2] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 481–490.

[3] M. Abdellatif, G. Hecht, H. Mili, G. Elboussaidi, N. Moha, A. Shatnawi, J. Privat, and Y.-G. Guéhéneuc, "State of the practice in service identification for soa migration in industry," in *Service-Oriented Computing*. Springer International Publishing, 2018, pp. 634–650.

[4] R. Khadka, A. Saeidi, S. Jansen, and J. Hage, "A structured legacy to soa migration process and its evaluation in practice," in *2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2013, pp. 2–11.

[5] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023.

[6] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, "A taxonomy of service identification approaches for legacy software systems modernization," *Journal of Systems and Software*, vol. 173, p. 110868, 2021.

[7] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: issues and directions," *IEEE Software*, vol. 16, no. 5, pp. 103–111, 1999.

[8] A. A. Almonaies, J. R. Cordy, and T. R. Dean, "Legacy system evolution towards service-oriented architecture," in *International Workshop on SOA Migration and Evolution*, 2010, pp. 53–62.

[9] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: migration and architecture smells," in *Proceedings of the International Workshop on Refactoring*. Association for Computing Machinery, 2018, p. 1–6.

[10] S. Johann, " Dave Thomas on Innovating Legacy Systems ," *IEEE Software*, vol. 33, no. 02, pp. 105–108, 2016.

[11] C.-Y. Li, S.-P. Ma, and T.-W. Lu, "Microservice migration using strangler fig pattern: A case study on the green button system," in *2020 International Computer Symposium (ICS)*, 2020, pp. 519–524.

[12] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909.

[13] M. Razavian and P. Lago, "A survey of soa migration in industry," in *Internation Conference on Service-Oriented Computing*. Springer, 2011, pp. 618–626.

[14] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2020.

[15] L. Bao, C. Yin, W. He, J. Ge, and P. Chen, "Extracting reusable services from legacy object-oriented systems," in *International Conference on Software Maintenance*, 2010, pp. 1–5.

[16] Z. Zhang, H. Yang, and W. C. Chu, "Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration," in *International Conference on Quality Software*, 2006, pp. 385–392.

[17] M. L. Brodie and M. Stonebraker, "Darwin: On the incremental migration of legacy information systems," *Distributed Object Computing Group, Technical Report TR-0222-10-92-165, GTE Labs Inc*, vol. 28, 1993.

[18] M. Fowler, "Strangler fig," 2004, accessed: 2024-10-24. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html

[19] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018.

[20] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans, "On the definition of service granularity and its architectural impact," in *Advanced Information Systems Engineering*. Springer, 2008, pp. 375–389.

[21] J. Liu, Y. Li, T. Tan, and J. Xue, "Reflection analysis for java: Uncovering more reflective targets precisely," in *International Symposium on Software Reliability Engineering*, 2017, pp. 12–23.

[22] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse, "Test case selection in industry: An analysis of issues related to static approaches," *Software Quality Journal*, vol. 25, pp. 1203–1237, 2017.

[23] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Software*, vol. 35, no. 3, pp. 63–72, 2018.

[24] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: Workshop on Dynamic Analysis*, May 2003, pp. 24–27.

[25] A. Utture, S. Liu, C. G. Kalhauge, and J. Palsberg, "Striking a balance: pruning false-positives from static call graphs," in *International Conference on Software Engineering*, 2022, p. 2043–2055.

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, p. 264–280.