SPReaD: Service-oriented Process for Reengineering and DevOps - Developing Microservices for a Brazilian State Department of Taxation

DA SILVA, Carlos <http://orcid.org/0000-0001-9608-439X>, JUSTINO, Yan and ADACHI, Eiji

Service Oriented Computing and Applications
https://doi.org/10.1007/s11761-021-00329-x

ORIGINAL RESEARCH PAPER

SPReaD: service-oriented process for reengineering and DevOps

Developing microservices for a Brazilian state department of taxation

Carlos Eduardo da Silva[1] · **Yan de Lima Justino**[2] · **Eiji Adachi**[3]

Received: 29 September 2020 / Revised: 2 September 2021 / Accepted: 19 September 2021

Abstract
The reengineering of systems into a microservice-based architecture can be seen as an implementation of a service-oriented architecture (SOA). However, the deployment of SOA into an enterprise is a challenging task, as it may involve the modernization of mission-critical systems with high technical debt and high maintenance costs. To this end, a process is required to provide an appropriate set of steps and techniques that minimize risks and at the same time ensure the quality of the systems during the migration process. Thus, this work presents the Service-oriented Process for Reengineering and DevOps—SPReaD, an instantiation of the mainstream SOA methodology focusing on the reengineering of legacy systems integrating DevOps aspects for developing microservices systems. This process has been defined during a real software reengineering project of legacy systems from a Brazilian State Department of Taxation. The results obtained include a substantial improvement in the quality of the main taxation system used by the state, including not only code-related metrics but also performance improvements of the services offered, and a change in the methodology adopted by the software development team.

Keywords Microservices · SOA · Software Reengineering · DevOps

1 Introduction

Microservices can be understood as a software approach and system architecture built upon well-established concepts of modularization and technical boundaries [14]. It has been the choice of development teams eager to adopt emerging practices whose design, standards, and technologies favor more streamlined software delivery. Also, microservices have aroused interest in teams that are eager to modernize their legacy monolithic systems [21,22]. It is possible to notice an increase in the number of reengineering projects focused on the migration of legacy systems to a service-based solution [1,24].

Although Service-Oriented Architecture (SOA) is considered as a heavy and complex solution for creating software systems, microservices share the same SOA design principles [21,25]. A *Service* is a software published via an API, which is part of a contract and provides a collection of resources [6]. These resources are grouped into logical units that represent a functional context. In fact, microservices do not constitute a new architectural style different from SOA, but qualify an implementation of SOA with state-of-the-art software engineering practices [31].

One of the main challenges to obtain a successful SOA project is in understanding how they should be carried out [6]. The *Mainstream SOA Methodology* (MSOAM) provides a generic reference model of SOA delivery. It is characterized by project and lifecycle stages represented by 11 phases, contemplating from initial planning, services analysis and design, development and testing, deployment and maintenance, to service versioning and retirement.

According to Fowler [10], "*with the microservice architecture, an application can be easily scaled horizontally and vertically, productivity and developer speed increase dra-*

✉ Carlos Eduardo da Silva
c.dasilva@shu.ac.uk

Yan de Lima Justino
contato@yanjustino.com

Eiji Adachi
eijidachi@imd.ufrn.br

[1] Department of Computing, Sheffield Hallam University, Sheffield, UK

[2] XP Inc, São Paulo/SP, Brazil

[3] Digital Metropolis Institute, Federal University of Rio Grande do Norte, Natal/RN, Brazil

Published online: 20 October 2021

matically, and old technologies can easily be switched to the newer ones". To that end, it is necessary to adopt continuous practices, that is, continuous integration, delivery, deployment, and monitoring to enable organizations to release new resources frequently and reliably, ensuring the high quality of the system deployed throughout its life cycle [3]. In industry, these continuous practices are labelled as DevOps, *a set of practices designed to reduce the time between making a change in a system and the change being put into production, while ensuring high quality* [3]. These practices together help shape decisions about how to build microservices and how to deploy them.

It is noteworthy that while microservices and DevOps reflect philosophies that refer to a structure of small independent teams, most organizations have a large, mission-critical system that is not designed in this way. These organizations need to decide whether they want to migrate their architectures to microservice architectures and which ones to migrate [3]. However, it is recognized that deploying a service-oriented solution is unique within each organization and may consider using different [6] approaches, depending on the nature and scope of the project. In addition, the degree of abstraction and complexities involved in reengineering a legacy system requires process instances that enable methodologies such as MSOAM to be adopted in conjunction with modern approaches such as microservices and continuous practices such as DevOps.

In this context, it is possible to identify a major design challenge in adopting microservices as a tactic for migration of legacy systems: the absence of a software reengineering process that leads to the migration of a monolithic system to a service-oriented solution in modern approaches such as microservices.

Based on this, the main contribution of this article is a process for the reengineering of legacy systems into a SOA-based system, employing microservices as implementation tactic. The *Service-oriented Process for Reengineering and DevOps* (SPReaD) is an instantiation of the MSOAM with a focus on a software reengineering process. SPReaD provides guidance for the development of service-oriented solutions based on microservices while employing DevOps techniques for continuous software delivery and monitoring.

The SPReaD process has been initially identified in a project conducted between 2016 and 2018 at the State Department of Taxation of *Rio Grande do Norte* (*SET-RN*)[1] during the migration of *UVT* (Taxation Virtual Unit) system, the main system of *SET-RN*. SPReaD has been presented as a poster at the 2018 ICSE's Software Engineering in Practice track [18]. Compared to that first presentation, this article presents a detailed description of the SPReaD process, its different artefacts, concrete details of its application. This article also reports on the use of the SPReaD process in a real project, with significant results regarding the achievement of important project goals in terms of software quality, and an impact on the software development and operations practice of *SET-RN*.

The remain of this article is organized as follows: Section 2 presents the *UVT* system as the target of the reengineering project and main motivation for the SPReaD process. Section 3 describes the SPReaD process as an instantiation of the MSOAM. Section 4 presents the use of the SPReaD process in the migration of the *UVT* system. Section 5 presents an evaluation of our approach. Section 6 discusses some related work, while Section 7 concludes the article.

2 The legacy UVT system

The SPReaD process was conceived and employed in the context of the project that reengineered the Virtual Unit of Taxation (*UVT*) system.[2] The *UVT* system is the main system supporting the business processes of the Department of Taxation of Rio Grande do Norte State (*SET-RN*) in Brazil. It was created between 2008 and 2010 to unify the business processes of the *SET-RN* which until that time were supported by small and independent applications.

The *UVT* system was structured as a single Web application and database, with the goal of providing all the services available to taxpayers, accountants, carriers and other actors who interact with the State Department of Taxation. The Web application had three main components: (i) View, (ii) Model and (iii) Middle-tier. The Model component represents the different entities of the system. The View component was responsible for implementing the presentation layer, that is, the user interface elements, while the Middle-tier component implements business logic. Each business functionality is encapsulated in a Web page composed of the View and Middle-tier components.

Over the years, the *UVT* system began to present problems compromising its evolution. The presentation logic and the business logic of the system were tightly coupled, hindering the reuse of functionalities, which ultimately promoted code cloning practices. The business entities were poorly modularized, with the same entity improperly accumulating responsibilities of different domain contexts of the system. This often led to ambiguities about the identity of entities and undesirable side effects caused by changes in apparently unrelated features. The *UVT* system also made extensive use of synchronous communication, which slowed system performance when long processing tasks were required.

In order to deliver any new functionality, or corrective maintenance, the entirety of the *UVT* system needed to be

[1] *Secretaria de Estado da Tributação do Rio Grande do Norte.*

[2] *Unidade Virtual de Tributação, in Portuguese.*

fully compiled. This was a completely manual and complex process with few quality control activities. That is, even a small, isolated adjustment forced the entire system to be recompiled and redeployed. Also, the absence of automated source code integration mechanisms to deal with the conflicts generated during development reduced the quality of the resulting code, favoring more frequent bugs in the production environment, with negative and unwanted side effects.

The problems presented above harmed the system's ability to respond to changing demands. Moreover, requests for renewing the visual aspects of the system and specially the necessity of providing *UVT* services to mobile devices intensified the need to modernize it. Given the accumulation of problems over the years and the difficulty of promptly meeting new demands, the *SET-RN* decided to abandon this version of the *UVT* system and completely reengineer it.

In this context, *SET-RN* started a reengineering project to modernize the *UVT* system. Due to the lack of a well-defined process to govern the migration of the legacy system, in this project, SOA was adopted as a strategic paradigm for modeling the desired solution whereas microservices were adopted as an implementation tactic. In addition, during the development of the new version of the system, DevOps practices were adopted for integrating, delivering, deploying and monitoring the system. And it was from the combination of these concepts that SPReaD was conceived.

3 The SPReaD process

The Service-oriented Process for Reengineering and DevOps (SPReaD) is an instantiation of the Mainstream SOA Methodology (MSOAM) [5,6]. MSOAM is a reference model for SOA project delivery, providing a set of generic activities and practices that needs to be customized for its usage. We organized the MSOAM activities on a reengineering process based on the software engineering process framework described by Pressman and Maxim [23]. Their framework identifies five generic phases applicable to all software projects: (i) Communication, (ii) Planning, (iii) Modeling, (iv) Construction and (v) Deployment. In the context of SPReaD, the Modeling phase is divided into *Analysis* and *Design*, the Construction phase is divided into *Code* and *Test* and the Deployment phase is divided into *Delivery* and *Support and Feedback*. The diagram in Fig. 1 presents an overview of the SPReaD process structure; the MSOAM activities are represented as blue rectangles.

SPReaD, as depicted in Fig. 1, comprises a Software Reengineering Process followed by a DevOps process. The rengineering part of SPReaD consists of a Reverse Engineering phase followed by a Forward Engineering phase. The Reverse Engineering phase corresponds to the Analysis activity, whereas the Forward Engineering phase comprises the Design, Code and Test activities. Moreover, the DevOps part of the process consists of a *CI + CDE + CD* phase followed by a Monitoring phase, corresponding, respectively, to the Delivery activity, and to the Support and Feedback activity.[3]

The phases of SPReaD described above are applied iteratively to plan, model, build, and deliver one or more service inventories, which determine the scope and goal of a given iteration. In the next sections, we further detail the Modeling, Construction and Deployment phases of SPReaD. Due to space limitations, the Communication and Planning phases will not be covered in this article.

3.1 Modeling

The Modeling phase comprises the *Analysis* and *Design* activities. We further detail these activities next in Sections 3.1.1 and 3.1.2. The Design activity is the begin of the Forward Engineering part of the reengineering process; it comprises the *Service-Oriented Design* and *Service Logic Design* activities.

3.1.1 Analysis

In the context of SPReaD, the analysis activity corresponds to the Reverse Engineering part of the Software Reengineering process; it includes the *Service Inventory Analysis* and *Service Oriented Analysis* activities. Figure 2 details these activities.

The *Service Inventory Analysis* activity is responsible for producing the conceptual definition of service inventories. These inventories are documented in the *Service Inventory Blueprint* artefact, which describes main services of the target system. In the context of SPReaD, the *Service Inventory Analysis* comprises three main practical steps: (i) Identify Bounded Contexts, (ii) Identify Service Candidates, and (iii) Apply Context Mapping.

In the first and second steps, we employ the *Bounded Context* technique borrowed from Domain-Driven Design (DDD) [28]. Bounded context is used in DDD to isolate a domain-specific responsibility on an explicit interface boundary which decides which models to share with other contexts [20,21]. This technique is employed to delimit the applicability of certain models and keep them logically unified and consistent. It is applied over the business services identified in the legacy monolithic application, i.e., the different Web pages (view and middle-tier components) and models of the legacy system. These are then labelled and conceptually isolated into bounded contexts. In this process, legacy services and entity models that contain multiple

[3] In this context, CI, CDE and CD correspond to Continuous Integration, Continuous Deployment and Continuous Delivery, respectively.

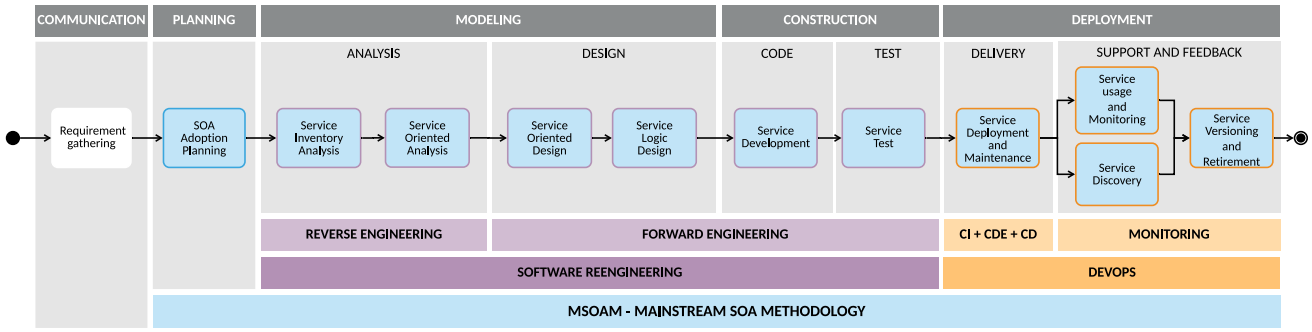

Fig. 1 General view of the SPReaD process

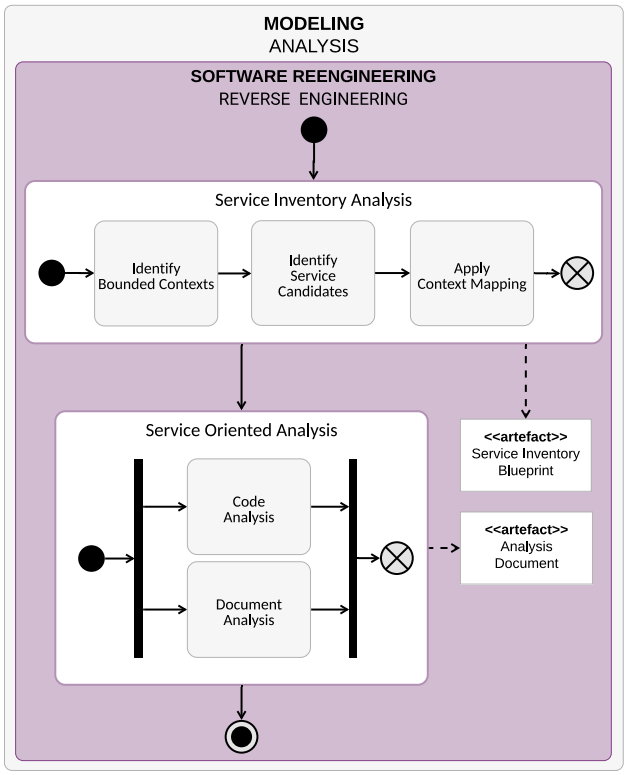

Fig. 2 Detailed view of service inventory analysis and service-oriented analysis activities of the modeling phase

responsibilities are separated into distinct elements. In the end, each bounded context corresponds to one candidate microservice, where each service of the legacy system corresponds to a capability to be offered by the identified microservice. These are then organized into *Service Inventory Blueprints* indicating the main domain contexts. An inventory may contain one or more bounded contexts depending on the inventory's business scope.

In the third step, we employ another DDD technique called *Context Mapping*. In a monolithic architecture, the communication between the components of the legacy system can happen unrestrictedly, as they all share the same core. Since we are decomposing this core in separate bounded contexts that are distributed into services, it is necessary to devise communication strategies between them, which can take different forms. Thus, we employ *Context Mapping*, which performs an integration pattern [28] to classify the relationship between external contexts and internal contexts.

The *Service Oriented Analysis* activity is responsible for creating new software documentation (Class Diagrams, Relational Entity Diagrams, Use Cases, etc.) to support the Forward Engineering process. This documentation helps explaining the scope of each inventory and its candidate services. The activities associated with analysis follow a set of practices, which are described in the sequence.

Having identified inventories and their candidate services, the practice to be adopted in service-oriented analysis is the beginning of the decomposition of the legacy service. This decomposition occurs by extracting the service logic from the Web page components (View and Middle-tier) in two focuses of analysis: one focus to identify behaviors and models that satisfy the logic associated with the graphical interface; and the other focused on service logic, which identifies the capabilities and business entities of the service. This results in artefacts as user interface prototypes, and service contracts and design. These are then used for the re-specification, re-design, re-architecture and re-codification of new user interfaces and service logic.

3.1.2 Design

Figure 3 illustrates the flow of activities from *Service Oriented Design* and *Service Logic Design*. Service-Oriented Design aims to re-design the software solution with a focus on reuse, as well as redefining the software architecture to express the new design features. In support of re-design, we adopt the service layer pattern [5] where services are categorized as: a) entity services, which is a reusable service with an agnostic functional context associated with one or more related business entities; b) task service, which is a non-agnostic functional context that generally corresponds to the logic of the single-purpose parent business process;

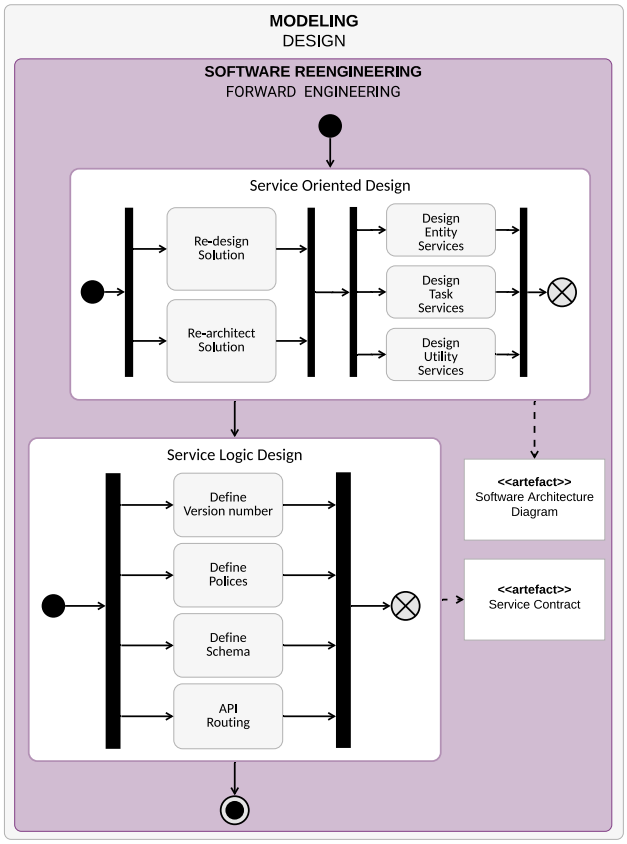

Fig. 3 Detailed view of service-oriented design and service logic design activities of the modeling phase

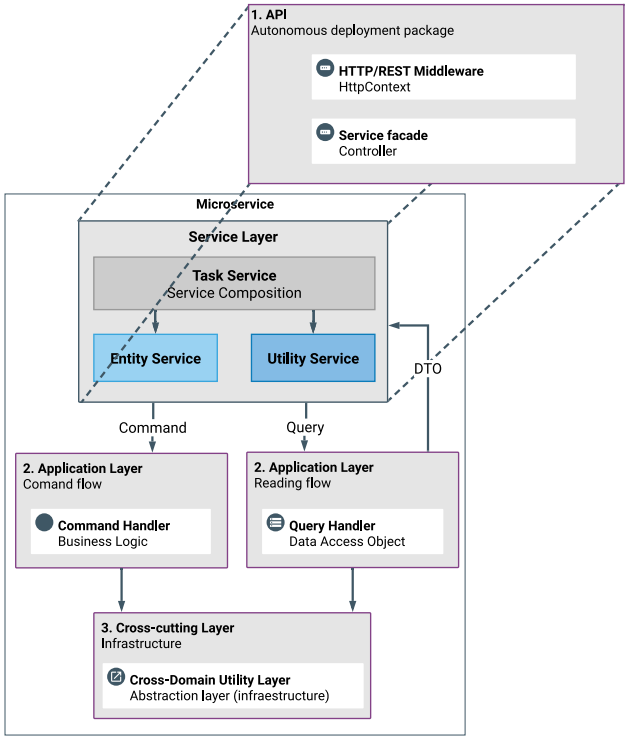

Fig. 4 General design structure for microservices

c) utility service, which is also a reusable service with an agnostic functional context, but this type of service is intentionally not derived from specifications and business analysis models.

In turn, the activity of Service Logic Design is responsible for completing the service logic, by defining a set of information such as the version of the service, its access policies, contract schemes and routes for accessing APIs, etc.

The result of *Design* is the specification of the service architecture components in terms of its modeling through structural diagrams, such as UML component and class diagrams. In addition, the set of metadata created will allow the construction of service contracts, which will facilitate the discovery of its capabilities.

SPReaD follows a design approach based on service layers and micro-task segregation. Figure 4 exposes the definition of what a microservice structure should look like. It is composed of an API layer that is responsible for handling HTTP/REST requests, including authentication and authorization in accessing resources, through facades to orchestrate the service logic. The API layer contains façades based on the service layers pattern. The application layer adopts the CQRS standard approach (*Command Query Responsibility Segregation*) [7], to separate the application into two flows: a writing stream called *Command*, in which we can find domain models and services to deal with business logic; and a *Query* flow, in which data transport objects (DTOs) that correspond to the service consumers' query operations reside. The *cross-cutting* layer provides a set of *cross-domain* components to abstract infrastructure resources, as well as to deal with the state maintenance of service objects.

Based on this model, the implementation of each microservice can be standardized to reflect a product with greater autonomy. In addition, the reduced and specialized scope of each microservice, as well as better componentization of the application layers and abstraction of the *cross-cutting* layer favor its portability and maintainability. Finally, by transferring the state of objects to infrastructure layers and adopting strategies for separating the application flows in writing and reading by adopting CQRS, the system gains a potential to achieve wide horizontal scalability and significantly improve its performance.

3.2 Construction

This phase involves the activities of *code* and *test*, which are related, respectively, to the activities of *Service development*, in which the actual service is implemented; and *Service test*, where software testing techniques are employed. They

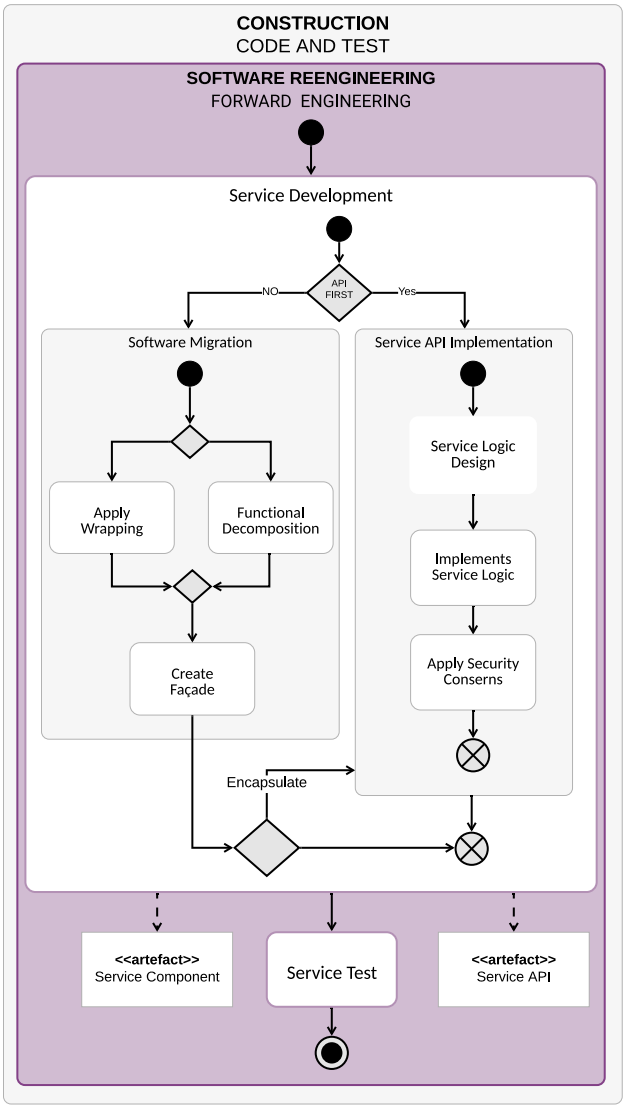

Fig. 5 Detailed view of service development activity of the construction phase

complete the forward engineering cycle within the software reengineering process.

Figure 5 details the flow of the service development activity. SPReaD is based on two approaches for service implementation: *Service API Implementation* and *Software Migration*.

The first approach, *Service API Implementation* concerns the development of the service contract defined by the *Service Oriented Design* in a scenario where the desired capability is not provided by an existing component of the legacy system. Based on the service contract, the service logic is designed and implemented, followed by the application of security concerns over the developed service.

The *Software Migration* activity is applied to legacy software in order to transport the service to a new software structure. For this, two techniques can be applied: the functional decomposition or wrapping [29]. Functional decomposition is used to extract entities and service logic from the legacy software. Those are encapsulated into business components that correspond to the designed service architecture. The wrapping technique is used to encapsulate legacy software with high complexity and an associated high migration cost. These usually involve shared resources, classes for infrastructure access, or external services. Finally, access to the migrated components is regulated by the creation of a service facade, which hides all the complexity in the software.

In some cases, the service contract needs to be made available before the service logic itself. For example, external teams hired to develop the front-end may demand access to the contract to proceed with the construction of GUI's (Graphic User Interfaces). In this scenario, the Web API is made available before the construction of the business component.

The SPReaD process does not define any specific step for the *Service Test* as different approaches can be employed according with the practices of the development team and the resources available. The use of unit tests is the basic recommended approach, with integration testing becoming necessary as the number of services deployed grows. This is also a situation that might require the availability of Web API before the construction of business components. Finally, black-box testing is used with those services that encapsulate legacy components.

3.3 Deployment

The deployment phase involves the activities *delivery* and *support and feedback*. They are responsible for managing the deployment of services into production and their monitoring. The delivery activity is based on the MSOAM's activity of *service deployment and maintenance*, while support and feedback is based on MSOAM's *service usage and monitoring*, and *service discovery*.

3.3.1 Delivery

The delivery activity corresponds to the service deployment and maintenance phase of MSOAM and explores the use of DevOps technique (continuous integration, continuous deployment, and continuous delivery). Figure 6 details this activity.

It starts by performing a continuous integration pipeline, which includes the integration of source code followed by its compilation (build application), automated tests and code analysis. These create reports containing integration and compilation information, indicators regarding test coverage and calculated technical debt.

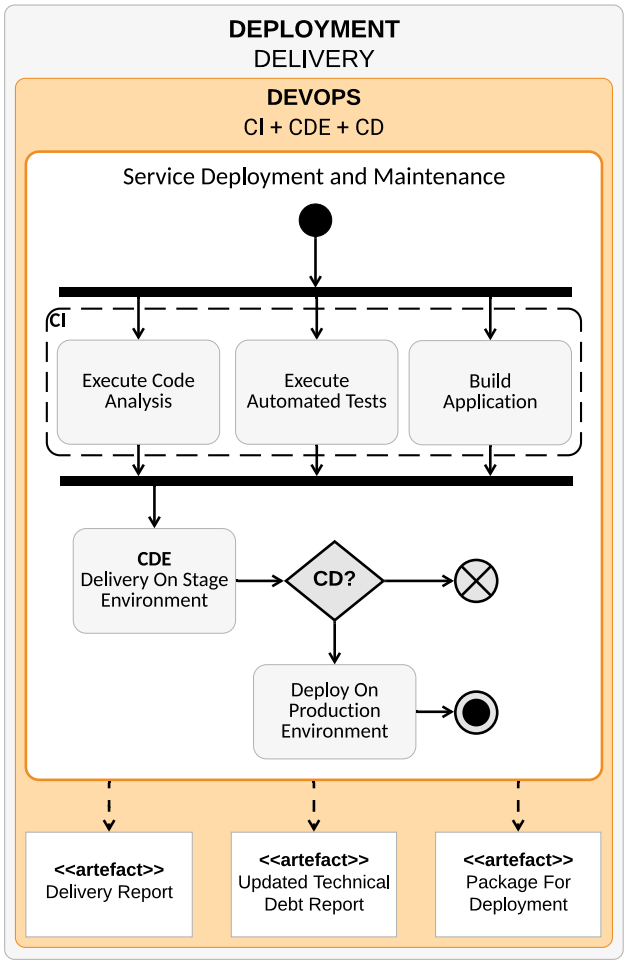

Fig. 6 Detailed view of service deployment and maintenance activity of the delivery phase

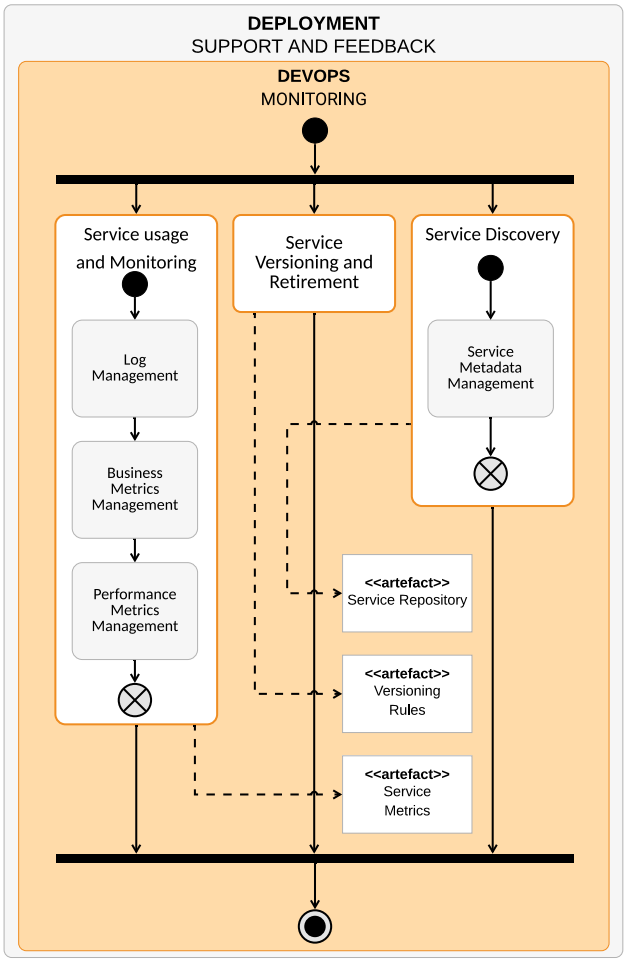

Fig. 7 Detailed view of activity of the deployment phase

When those activities succeed, the resulting package is delivered to an homologation environment (continuous delivery). In case continuous deployment practices are in place, the resulting package is deployed on a production environment. Otherwise the deployment in production is a manual process and thus this phase of the process is considered as ended.

3.3.2 Support and feedback

The *support and feedback* activity, detailed in Fig. 7, is mainly concerned with services that have been deployed and used in production environment. It includes MSOAM's *service usage and monitoring*, *service discovery*, and *service versioning and retirement*.

Service usage and monitoring is concerned with the collection, storage, processing, and visualization of different information about the services. It can include metrics used for performance, billing, scalability, and business logic-related information, such as logging. These are used by operations teams as means of monitoring the supporting infrastructure and by management teams to understand business dynamics through service consumption.

Service discovery concerns the provision of a repository, i.e., a service registry, where all services from the different inventories can be discovered. In essence, such repository allows the registration and querying of service metadata exposing the business capabilities and details of the active service instances, such as services contracts and endpoints.

The last activity, service versioning, and retirement, is responsible for supporting changes in existing services, such as changes to its logic or contract, with minimal disruption to existing service consumers. This requires version control tools and mechanisms, as well as tracking. Regarding versioning, the use of specifications such as Semantic Versioning plus the adoption of code version control tools such as Git can assist in managing service changes. With regard to tracking, the use of Service Catalogs in addition to the

use of monitoring tools such as the ELK stack[4] can indicate whether certain services continue to be used and by which customers, facilitating the planning of withdrawing services without consumption.

4 Application of the SPReaD process

This section presents the concrete application of the SPReaD process, using the reengineering of the *UVT* system as a concrete case study. We describe the different technical approaches adopted to achieve the objectives set by *SET-RN*guided by the phases of the SPReaD process. Given its relevance and being a central area of *SET-RN*, we will use the migration of *Tax Collection* services as an example to demonstrate the execution of these phases.

4.1 Modeling

During the modeling phase, the analysis focused on the definition of the *Tax Collection Inventory* and in the reverse engineering of its candidate services. Based on this, the design stage comprised the re-architecture and remodeling the candidate services. We detail each of these steps and their activities in the sequence.

4.1.1 Analysis

At the start of the application of SPReaD, all legacy services were within the same core without pre-established boundaries. Thus, the activity of *Service Inventory Analysis* was executed in order to delimit the business contexts based on the use of DDD.

For example, we have conducted a reverse engineering of legacy services *Tax Debt Management* and *Tax Payment Management*, which have been identified as the business capacities that are part of the *Tax Collection Management* context. The Tax Debt Management legacy service was composed by several pages, each providing a different business service, with one for each type of tax and for some auxiliary functionalities. All the pages had code-behind components that accessed the same Debit entity. We also noticed that this entity accumulated broad responsibilities, since it knew all the contexts of each debt origin, as well as the structures of relational mapping with the database. The same process has been conducted with the legacy services *City Halls Management* and *Resource Transfer Management*, which have been grouped in the *Resource Transfer Management* context.

[4] Elasticsearch, Logstash, and Kibana, a tool for collecting, analyzing and visualizing logging and performance information from different sources.

In our approach, each service inventory can have several limited contexts, which are implemented as a microservice. For example, the *Tax Collection Inventory* contains two microservices: *Tax Collection Management* and *Resource Transfer*.

As previous mentioned in Sect. 3.1.1, the communication between bounded contexts is mapped using another DDD approach called *context mapping*, which performs an integration pattern [28] to classify the relationship between external contexts and internal contexts. For example, the communication of the *Resource Transfer Management* microservice with the services of the external domain *City Hall* is intermediated by an *Anti-Corruption Layer (ACL)*, since the models of external requisitions of the city hall need to be adapted to models aligned to the *SET-RN* domain. The *Tax Collection Management* communicates with external domains as banks through the *Consumer-Supplier* strategy, sending (*upstream*) and receiving (*downstream*) documents following the communication specifications established between *SET-RN* and the banking institutions.

4.1.2 Design

In order to deal with the complexities identified in legacy services, each microservice was designed to offer an API and business components that have the service logic and entities migrated directly from the legacy system. These components encapsulate the different business capacities of the microservice. For example, as illustrated by Fig. 8, among the business capacities of the *Tax Collection Management* microservice there are *Tax Debit Management* and *Tax Payment Management*. One of the services offered by the *Tax Debit Management* component, for example, is *Bank Slip Handler* service witch orchestrates the logic flow, entity models (*Debit*) and business rules involved with bank slip generation.

During the migration of these components, some service logic needed to be wrapped in order to isolate code from the legacy system. This is the case of the *Bank Slip's File Transfer*, as software structures involved in this logic were already validated and approved by the banking institutions.

One of the challenges of software reengineering is dealing with legacy database entities. A decision on whether to migrate those entities into separate contexts must take into account the costs associated with this migration, as well as the cost in maintaining the legacy structure. Taking as an example the *Tax Debit Management* business component, the decision was to map the new components of the migrated services to the relational entity of the legacy database. The approach adopted is based on the use of façades, where each façade captures one business service and its respective model, i.e., one page of the legacy service. Thus, the legacy entity is decomposed into different façades and their respective mod-

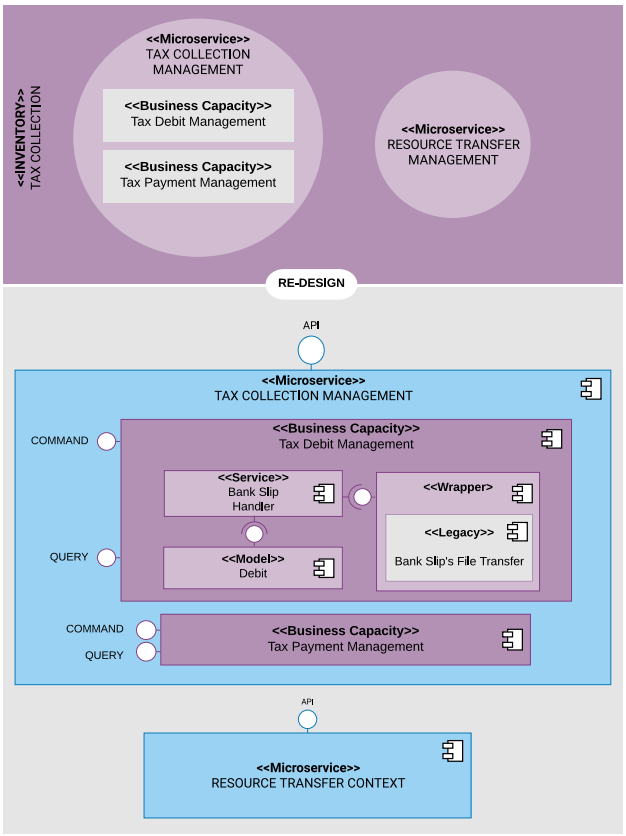

Fig. 8 Service redesign strategy applied to migrate the tax collection management services into microservices architecture

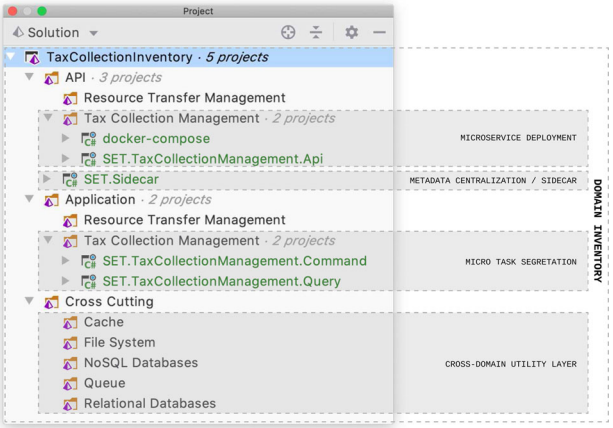

Fig. 9 Example of solution template for Visual Studio for structuring services source code and identifying the respective patterns used

els, which in turn have been grouped into different business services that are exposed by the *Tax Collection Management* microservice.

The different models encapsulated in the Tax Debit Management component have the attributes and business logic specific to its execution, as well as maps only the fields of the legacy database that they need. Thus, in the case of the migration of the *UVT* system, this was the limit to which software reengineering could be applied, which required a great effort on the part of the development team to deal with the discrepancies generated between the migrated models and the entities of the database. In spite of this limitation, this mapping may possibly be revisited in the future in order to define strategies for migrations that involve the reengineering of the database.

4.2 Construction

As previously mentioned, the construction phase encompasses the *service development* activity where the actual code of the new services is produced, and *service test* which deals with the testing of the produced services.

Each service inventory is built as a standalone software solution using .Net Framework, C# language and Visual Studio IDE Solution Templates. These templates are based on a set of patterns from the SOA Project Pattern Catalog [5] and provide programmers with a starting point on how to structure the source code of services. This allowed software projects and libraries to be organized to represent the delimited contexts of the domain inventory pattern. Figure 9 presents an example of the solution template for the Tax Collection inventory, which contains two microservices: *Resource Transfer Management* and *Tax Collection Management*.

A solution is divided in three basic directories: API, Application and Cross Cutting. API contains the artefacts related to the service contract of each microservice (as presented in Fig. 4), which will be detailed further on this section, and configurations for stand-alone deployment of microservices. The microservice API follows the microservice deployment pattern, in which each microservice is a self-compiling project that encapsulates business components and other dependencies in publishing packages that can be distributed, containerized, and replicated to web application servers such as IIS. We also employ the container sidecar pattern. Microservices usually need to access utility components that provide services such as monitoring and logging. However, these components are not an internal part of the microservices, requiring some level of isolation. The sidecar pattern provides a structured way of deploying these utilitary components in isolated containers that run in the same hosts, reducing the cost of a remote communication. The application directory contains the business logic implementation of the microservice according to the micro-task segregation pattern, an application of the CQRS approach. These are identified in Fig. 9 by *SET.TaxCollectionManagement.Command* and *SET.TaxCollectionManagement.Query* directories. The Cross-Cutting directory implements the cross-domain utility layer, containing the code for accessing services like cache, file system, queue and databases.

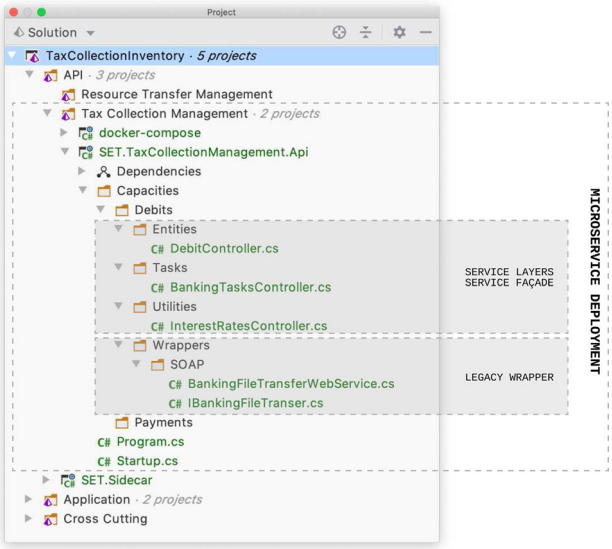

Fig. 10 Detail of the service layer implementation of the tax collection management microservice

Figure 10 presents the elements of the API implementation of the Tax Collection Management microservice. This part of the code is structured based on the service layer pattern, where the services are organized into Entity, Task and Utility layers. Each service then is implemented as a controller (e.g., DebitController) acting as a façade that abstracts the actual service logic implementation, and defining a clear entry point for its clients, avoiding the coupling of the service logic with the presentation layer. The controllers are implemented using the Asp.Net Web API framework. Legacy services are encapsulated and their calls in are brokered by a contract, which extracts, encapsulates, and eliminates technical details of the legacy component's logic. These are grouped inside the *wrappers*, which in this example, is abstracting away from a SOAP Web service (e.g., *BankingFileTransferWebService.cs*. Any access to a legacy component is achieved through an interface (e.g., *IBankingFileTransfer.cs*) as to prevent the new service from knowing implementation details of the legacy logic.

As previously mentioned, SPReaD does not prescribe any specific strategy for testing, instead recognizing that such activity must be included according with the resources available to the development team. In the case of the *UVT* system, we have employed unit testing during the coding phase in order to validate that the new business components performed as expected. Integration tests were also applied on microservices in order to validate their relationship with some infrastructure components such as database and messaging systems, as well as validate their composition with other network services. Finally, acceptance tests were also applied to ensure that the contracts set out in the APIs were in compliance with the specification.

4.3 Deployment

The deployment phase is composed by the *delivery*, in which we describe how DevOps continuous practices have been adopted; and *support and feedback* to support the monitoring, discovery and versioning of services in the production environment. We detail each of these steps and their activities in the sequence.

4.3.1 Delivery

The delivery phase concerns the infrastructure necessary for supporting DevOps continuous practices and the definition of the production environment of the target system.

The continuous integration pipeline is based on the TFS solution (Microsoft Team Foundation Server). This new scenario required the *SET-RN* team to better manage configuration to handle continuous code integration, coupled with automated testing, to ensure the quality of deployment of deliverable packages.

This set of tools helped *SET-RN* to centralize and better manage software configurations in the development, approval and production environments. It also assisted in monitoring the continuous integration process by providing quick notifications about the status of the software compilation to those interested in delivery. In addition, it provided software quality monitoring through the execution of automated tests and the survey of code metrics, which allowed the team to specify the interruption of software delivery with failures or below the desired quality expectations.

In order to support CD and CDE, we have created an stage environment corresponding to a clone of the production infrastructure, which is described in Fig. 11. Each microservice can be distributed into several network nodes, and replicated according with the need. These are then organized in two availability zones: production environment and contingency environment.

This has been possible through the removal of state management functionality from the service to cache components using the Redis tool. We employ the centralized isolated state repository pattern [7], in which microservices publish state information to a a master state repository. This centralized state database is deployed in its own containerized environment, capable of scaling as needed, and used for querying state information.

We employ a gateway architecture, where a unified point of entry is responsible for providing load-balance, security and managing routing to services. We have also identified cross-cutting components, such as database and identity man-

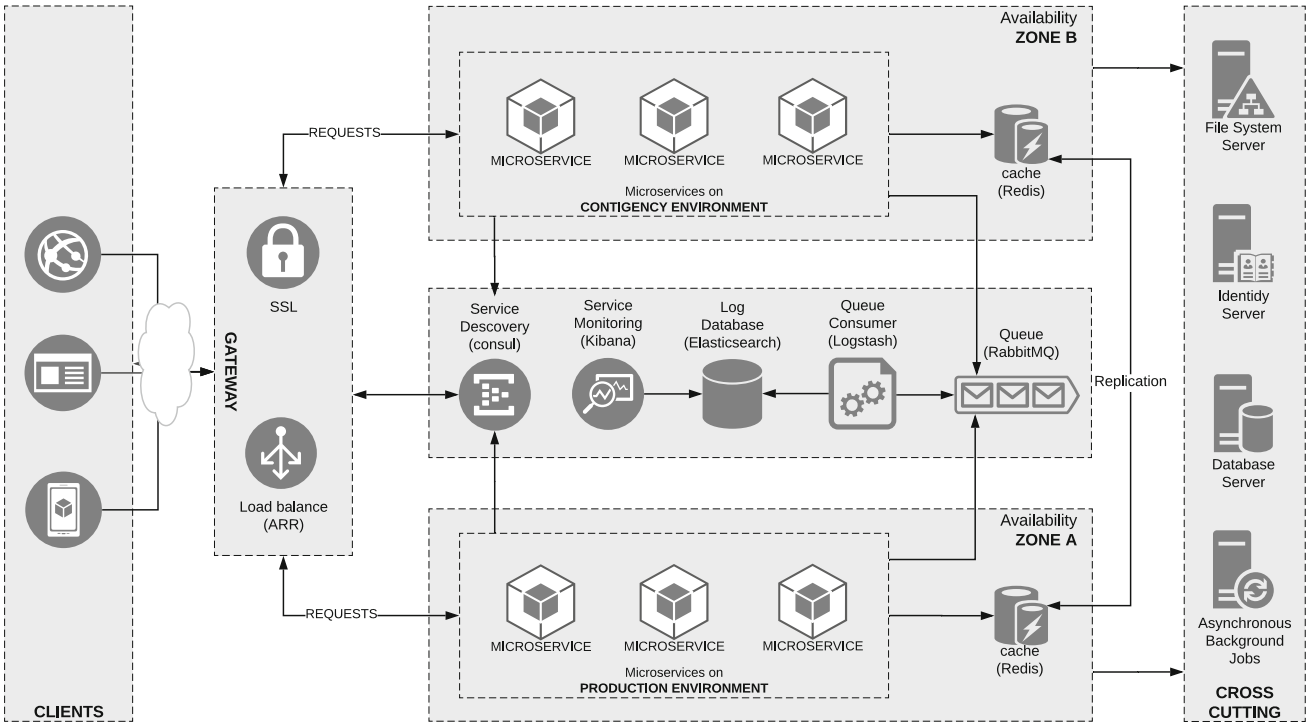

Fig. 11 General view of the supporting infrastructure of the UVT system

agement systems, that have been grouped into their own deployment environment.

There are a number of approaches for dealing with database design for microservices, ranging from a dedicated databases where each microservice has exclusive access to its individual database implementation, to shared isolated database deployed in its own individual container and independently managed. Since we are dealing with a legacy database, we employed the shared isolated database pattern.

4.3.2 Support and feedback

The activity of support and feedback is focused on monitoring of different aspects, such as service usage and monitoring; service discovery; and service versioning and retirement.

Regarding service usage and monitoring, tools were adopted to obtain data on the use of the services deployed in the production environment. The tools used for this purpose were RabbitMQ, which queued the log objects of the various instances of microservices; the ELK technology stack (Elastic search, Logstash and Kibana), an end-to-end solution that provides insights based on real-time monitored data and presented iteratively through dashboards. In this way, support teams can monitor the consumption of infrastructure resources, as well as for the management teams to understand the dynamics of the business through the consumption of services.

Regarding service discovery, a set of metadata was generated from the source code of the services contracts, which aided the formalization of service inventory contracts and the discovery of the capabilities associated with them. The service inventory metadata were generated from the OpenAPI specification by the Swagger tool[5]. These metadata are published together with our inventory and make up the *SET-RN* services repository portal. This portal provides a standard way of searching for services capabilities, being employed by the load balancing tool to discover services instance.

5 Evaluation

This section presents the results obtained by applying SPReaD for the modernization of the *UVT* system in order to evaluate our approach. We start by presenting some quantitative results and then discuss the results of adopting service orientation.

5.1 Results of Software Reengineering Applied to UVT System

As a result of software reengineering activities, applications previously coupled with the legacy *UVT* system are better isolated, decoupling the business logic of graphical

[5] https://swagger.io/specification/.

Table 1 *UVT* system technical debt before and after migration

Category	Before Effort	Cost	Fixes	After Effort	Cost	Fixes
Architecture	176d	70.6K	737	8d	3.21k	135
Code smells	51d	20.6K	273	22d	8.87k	301
Dead code	–	–	–	6h30min	325	37
Design	4d1h	1.65K	449	4d5h	1.86k	363
Immutability	17d7h	7.17K	889	6h32min	327	53
Naming conventions	1h25min	70.8	66	3d1h	1.29k	249
OOP design	8d5h	3.46K	837	2d4h	1.05k	610
Source organization	3h45min	15	188	7h	350	28
Visibility	2d4h	1.02K	656	4h22min	219	421
Total	**267d**	**107K**	**4217**	**48d**	**19.6k**	**2428**

interfaces. This represented greater portability, reuse and a significant improvement in software maintenance. This is noticeable when we analyze the technical debt of the migrated system using the "NDepend" tool [6].

Table 1 presents the comparative results of the technical debt between the legacy system and the migrated system in terms of effort (in days, hours and minutes), cost (in R$) and amount of fixes[7]. The results show a significant reduction in the technical debt of the migrated code. We can notice a 58% reduction in the amount of fixes required. The effort required to make all these corrections went from 267 days to 48 days, which meant a cost reduction from R$ 107K to R$ 19K.

We also registered a significant decrease in effort in the *Architecture* category from 176 days to 8 days. Metrics such as *Code Smells*, *Immutability*, *OOP Design* and *Visibility* also decreased in debt. This indicates that from a reengineering point of view the software migration managed to produce better quality code. On the other hand, *Naming Conventions*, *Source Organization* and *Design* metrics showed a relative increase. In addition to these metrics, there was also a record in the migrated code base of occurrences of the *Dead Code* category. This may indicate problems in managing migrated source code, mainly due to its granularity and decomposition.

For the purpose of comparing this granularity, Table 2 presents the *UVT* system LoC (Lines of Code) metrics prior to migration and the microservices created after reengineering process. In this table, the *Rating* column presents a sustainability rating scale related to the value of the technical debt ratio in the project. This classification is based on the relationship between the code base size and the estimated time to correct all identified issues. The rating "A" indicates that the pending correction cost is less than 5% of the estimated fix time, while a rating of "C" indicates that this cost is between 11% and 20%. The column #LOCs presents the number of lines of code.

Table 2 LoC metrics in legacy code Vs microservices code

Rating	UVT	#LOCs
C	Legacy Code	282K
Rating	Microservices	#LOCs
A	Tax Collection	43K
A	Invoices	39K
A	Taxpayer Registration	29K
A	Carriers	24K
A	Fiscal Inspection	41K
A	Service Schedule	11K
A	Document Authenticity	22K
A	Tax Statements	11K
A	Debt Installments	27K
		247K

Note that the legacy code of the UVT system is rated **"C"**, in turn, inventories received a rating **"A"** on the sustainability rating scale. This indicates a very low technical debt. It is also noted that the total number of LoC in Service Inventories (247K) approximates the legacy system LoC number (282K). However, as each Inventory has an independent maintenance cycle, this number drops considerably.

In addition to the code metrics, performance metrics of the deployed services were extracted from the production environment. Table 3 presents some service performance indicators obtained through the ELK monitoring infrastructure. These metrics were extracted in different moments between June/2017 and June/2019 and were grouped by total number of requests (#Requests) in the month, average response time (in milliseconds) of all services and number

[6] This choice was based on the fact that it is based on the SQALE method (http://www.sqale.org/details) and his analysis is made of the .NET framework Intermediate language (IL), thus covering all the languages of the family. .net framework (C#, Visual Basic, and F#) from the same analysis perspective (https://www.ndepend.com/).

[7] The effort is measured by the man–day ratio to develop 1000 lines of code. By default, the tool adopts the 18 man–day ratio, which means an average of 55 lines of code written per developer in one day.

Table 3 Service performance results

Month	#Requests	Response time (ms)	#Users
Jun/2017	197K	222	939
Jul/2017	484K	184	4,079
Aug/2017	652K	199	5,162
Sep/2017	3.8M	187	11,156
Oct/2017	4M	242	11,347
Nov/2017	4.2M	204	15,301
Dec/2017	4.4M	203	14,519
Jan/2019	6M	268	16K
Feb/2019	6M	264	16K
Mar/2019	6M	241	15K
Apr/2019	7M	247	18K
May/2019	8M	216	35K
Jun/2019	11M	200	71K

of users (# Users) answered. The *UVT* pilot project was officially launched in June/2017 for an initial user base of less than 1,000 users. In July/2017, the user base was substantially expanded (over 4,000 users), and the number of requests more than doubled (484,665 versus 197,458 from the previous period), while the response time remained around 200 milliseconds. In August/2017, there was a slight increase in the number of users (over 5,000 users) and the number of requests (652,455) with a stable average response time (199.21 milliseconds).

By August/2017, both the legacy system and the new system were active in parallel. However, in September/2017, the old system was deactivated and from that moment all users started consuming services on the new *UVT* system. Thus, we had a 116% increase in the user base (from 5,162 users to 11,156 users) and a 496% increase in the number of requests (from 652,455 requests to 3,889,923 requests).

This was a moment of apprehension for the team, but despite the huge increase in load, the average response time remained stable at around 200 milliseconds. In addition, the feedback received from the team was that there was a positive feeling about the overall system performance, as well as acceptance feedback from the new Web system graphical interface and mobile applications. In the following months, from October/2017 to December/2017, the system presented an increase of 557,712 requests and 4,145 users. However, performance remained close to 200 milliseconds.

It is important to mention that between June/2017 and December/2017, the operations team was actively monitoring the entire environment as part of the deployment phase activities. This has allowed us to identify infrastructure bottlenecks and make some proactive changes, such as scaling infrastructure resources allocated to databases and cache services, as well as adding new service instances.

During the first trimester of 2019, the system maintained stable numbers of requests, users and response time. In May/2019, there was the launch of a new mobile application for the general population, which provided a substantial increase in the number of users. Nevertheless, due to the DevOps and monitoring practices introduced by SPReaD the infrastructure was dynamically adjusted according with the load, maintaining an average response time between 200 and 250 miliseconds.

During the second semester of 2019, the SPReaD process continued to be applied and new services have been identified, developed and deployed into the infrastructure. However, we are no longer able to publicize performance indicators about the services running at *SET-RN*.

5.2 SPReAD adoption results

The application of SPReaD in the *UVT* project has provided *SET-RN* with substantial benefits, such as reaching the *Service Aware Level*, one of the maturity levels of service-oriented organizations [4]. By reaching this level, it has been confirmed that the relevant business requirements and goals are in place and that the global organizational foundation required for the SOA initiative is in place.

In addition, we also noticed benefits to the software development lifecycle adopted at *SET-RN*. For example, the adoption of a standardized service agreement practices allowed the creation of metadata that help in the formalization of service inventory contracts and the discovery of associated capabilities. This increased the alignment between IT and business, as the different stakeholders now have a foundation on which to locate agnostic services that can be used to define new business processes. This motivated auditors to redefine their requirements specification and project organization processes, which can be seen from the search for tools such as business process management (BPM) and the interest in collaborating with service organization by building a *SET-RN* service catalog.

Another noticeable benefit of the project was the capability of realtime service monitoring, which not only provided a pro-active and preventive approach to the support operations team, but combined with service metadata allowed an understanding of the business dynamics from the consumption of resources by the clients of the *UVT* system.

These results were presented to the business analysts and the *SET-RN* management team, with very positive reactions from the CODIN (Informatics Coordination) management and the State Department of Taxation. The team was invited by the Inter-American Development Bank (IDB) to present the *UVT* migration project as one of the outstanding success stories, due to the deadline and efficient resource management.

The impact of this project has transformed *SET-RN*'s way of planning and building software solutions. All this has an impact on the efficiency of the state of Rio Grande do Norte's tax collection. In addition, there are new demands for applying SPReaD to other *SET-RN* systems, and a new project is being conducted.

5.3 Lessons learned

The main lesson learned in this project is the fact that the construction of a large system using microservices still needs aspects from more traditional SOA. For example, during the first iterations of the migration project we have found some gaps in the modeling and construction of microservices that were filled by seeking in SOA a paradigm to guide the reengineering of legacy systems. The software quality results obtained provide evidence that the SPReaD process is able to follow SOA standard-based concepts for dealing with the construction and deployment of services, without neglecting modern aspects of microservices and DevOps-based solutions.

Working with a system that can be considered as critical, it is not always possible to completely migrate all legacy code in one shot. For example, some database structures cannot be easily changed; libraries and pieces of software that have been certified or approved by third parties must be maintained. Finally, it is necessary to understand that new technical debts can be raised during the process, in addition to those that were inherited through codes that were directly incorporated into the solution.

6 Related work

This section presents some work related to the topics discussed in this article, namely the migration of legacy systems and the use of MSOAM in development or migration projects. The main issue tackled by SPReaD is the lack of concrete guidance for the reengineering of legacy systems into microservices that encompasses all phases of the software development process.

Because of its flexible coupling characteristics, published interfaces, and a standard communication model, SOA allows legacy systems to be exposed as services [17]. However, any specific migration requires a concrete analysis of the feasibility, risk and cost involved. In this sense, strategic identification and service extraction from legacy code is also crucial.

Along these lines, articles that are relatively close to the beginnings of SOA, such as [24] and [27], consider the fundamental principles of orientation to service as evaluation elements for the adequacy of existing SOA assets, besides proposing metrics and guidelines that support the evaluation of these principles. The goal of these researchers is to help organizations understand the efforts involved in an SOA migration by assisting them in identifying services from existing assets. However, this approach is too abstract and does not point to process instances closer to service implementation.

On the other hand, some authors provide methods for guiding such migration [1,16]. For example, Baghdadi and Al-bulushi [1] propose the use of wrapping techniques to extend the business logic of legacy applications while preserving investments through their migration to services. However, by simply encapsulating legacy logic into service layers, we are fatally encapsulating its technical debt, which may lead to a rapid software decay. In this sense, the indiscriminately application of wrapping technique can anticipate the need for new migrations and bring in new associated costs that could have been better applied if employed in a reengineering process.

More recent works have dealt specifically with methods for the migration to microservice architecture. Escobar et al. [8] present an approach for migrating Java enterprise applications into a microservice architecture. Their approach performs source code analysis of Enterprise Java Beans grouping them into microservices based on data flow and method invocation between the beans. Lin et al. [19] describe a method for migration of Web applications into a cloud computing environment following a microservice architecture. However, the description is very abstract with generic instructions. Kecskemeti et al. [15] present the ENTICE project, a methodology for decomposing monolithic services into microservices. Their approach is focused on strategies for optimizing the construction and composition of services into virtual machine and/or container images.

Other works report on the migration experience. Gouigoux and Tamzalit [13] report on the experience of migrating a real legacy system into a microservice architecture. However, they focus on three aspects of the migration process: defining service granularity, service deployment and service integration. Balalaie et al. [2] report on the experience of migrating a real system to a microservice architecture. They define specific steps for the application under migration. Interestingly, they report on a series of lessons learned that indicates the need to a higher-level of guidance, such as the one provided by service-orientation, and considered by SPReaD, such as the need for service contracts and templates for service development.

Unlike previously mentioned proposals, our approach presents a process that deals with legacy systems through the application of software reengineering, whereby the logic of the legacy application is migrated using, in addition to the wrapping technique, functional decomposition. In addition, our approach takes into account the technical debt ratio and quality attributes required to meet the logic to be migrated.

Thus, the choice of the strategy to migrate a legacy application tends to be based not only on the need to expose a capacity via service. Moreover, by being developed in an industrial setting, the SPReaD technique not only adopts a holistic view but also incorporates those concerns that are usually not visible to academics.

In fact, a number of surveys and systematic studies have analyzed the microservice topic [11,12,14] with one of the most recent by Waseem et al. [30], at the time or writing this article, in which the focus was on analyzing the relationship between microservice architecture and DevOps. It is interesting to notice that the majority of works are focused on specific aspects of the microservice architecture, and do not provide a holistic view, considering the full software development life-cycle. Some of these surveys [12,14] have indeed identified future challenges that are incorporated into the SPReaD process, such as design (section that talks about design), granularity, implementation, deployment, and monitoring, with a strong link to the continuous practices of DevOps.

More closely aligned to our work is the approach of Fan and Ma [9], in which a migration process considering the full software development life-cycle is presented. However, their description is very superficial and does not provide enough details as to be adopted for a migration project besides mentioning the use of DDD techniques and some of the tools used. In contrast, the SPReaD process employes the MSOAM methodology as a solid foundation for guiding the reengineering process. Moreover, we recognize that not all migrations project will be able to deal with 100% of the target software, such as dealing with legacy databases.

More recent references to MSOAM include the work by Santika, Suhardi and Yustianto [26], in which they proposed the *Service Engineering Framework* derived from an approach that uses MSOAM-combined business analysis tools to avoid redundant activities between analysis of business processes and the SOA methodology. This framework consists of four phases (identification, design, development, and deployment) and was used to build services from a local government financial system (municipality). Although this research seeks alternatives to the implementation of SOA by using MSOAM as a methodological approach, this use boils down only to analysis, neglecting the other phases.

Unlike the approaches presented, our proposal can be seen as an instantiation of the MSOAM focused on software reengineering, integrating DevOps to deal with some phases of MSOAM Deployment. We also report a concrete instantiation of the proposal in a real project with millions of users. In addition, our process is in line with the demands of the *Service Oriented Architecture (SOA) Maintenance and Evolution Research Agenda* [17], specifically in the topics: Process and Cycle. Life, Architecture and Design, Deployment, Maintenance and Evolution, prepared by SEI (Software Engineering Institute).

7 Conclusion

In this work, we presented the *Service-oriented Process for Reengineering and DevOps* (**SPReaD**), a process for the reengineering of legacy systems into microservice architectures. We also presented a report of an industry project applying SPReaD to reengineer the *UVT* system, an information system maintained by the Rio Grande do Norte State Department of Taxation (SET). Results show that the adoption of SPReaD allowed SET to reach the *Service Aware Level* of the SOA Governance Model, promoting a better alignment between IT solutions and business process in the organization. Also, the reengineered version of the *UVT* system provides better quality attributes when compared to its legacy version, specially in terms of performance and scalability.

As it is being adopted in the context of other projects, the principles, standards and tools promoted by SPReaD are helping to unify and improve SET's software development process. In fact, some principles, standards and tools of SPReaD are being adopted in SET even in the context of projects not related to the reengineering of legacy systems. As future work, we intend to analyze how SPReaD is being adopted and adapted to these new projects, seeking opportunities for improving and evolving it.

References

1. Baghdadi Y, Al-Bulushi W (2015) A guidance process to modernize legacy applications for soa. Ser Orient Comput Appl 9(1):41–58. https://doi.org/10.1007/s11761-013-0137-3
2. Balalaie A, Heydarnoori A, Jamshidi P (2016) Migrating to cloud-native architectures using microservices: An experience report. In: Advances in service-oriented and cloud computing, pp. 201–215. Cham: Springer. https://doi.org/10.1007/978-3-319-33313-7_15
3. Bass L, Weber I, Zhu L (2015) DevOps: a software architect's perspective. Addison-Wesley Professional
4. Erl T (2010) SOA Governance. Prentice Hall
5. Erl T, Carlyle B, Pautasso C, Balasubramanian R (2012) SOA with REST: principles, patterns and constraints for building enterprise solutions with rest, 1st edn. Prentice Hall Press, Upper Saddle River, NJ, USA

6. Erl T, Merson P, Stoffers R (2017) Service-oriented architecture: analysis and design for services and microservices. Prentice Hall
7. Erl T, Naserpour A (2020) Microservice and containerization patterns. Arcitura Education Inc. https://patterns.arcitura.com/microservice-patterns
8. Escobar D, Cárdenas D, Amarillo R, Castro E, Garcés K, Parra C, Casallas R (2016) Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American computing conference (CLEI), pp. 1–11 . https://doi.org/10.1109/CLEI.2016.7833410
9. Fan CY, Ma SP (2017) Migrating monolithic mobile application to microservice architecture: an experiment report. In: 2017 IEEE international conference on AI mobile services (AIMS), pp 109–112. https://doi.org/10.1109/AIMS.2017.23
10. Fowler SJ (2016) Production-ready microservices: building standardized systems across an engineering organization, 1st edn. O'Reilly Media, Inc
11. Francesco PD, Lago P, Malavolta I (2018) Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE international conference on software architecture (ICSA) (2018). https://doi.org/10.1109/ICSA.2018.00012
12. Garriga M (2017) Towards a taxonomy of microservices architectures. In: International conference on software engineering and formal methods, pp 203–218. Springer . https://doi.org/10.1007/978-3-319-74781-1_15
13. Gouigoux JP, Tamzalit D (2017) From monolith to microservices: lessons learned on an industrial migration to a web oriented architecture. In: 2017 IEEE international conference on software architecture workshops (ICSAW), pp 62–65 . https://doi.org/10.1109/ICSAW.2017.35
14. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S (2018) Microservices: the journey so far and challenges ahead. IEEE Softw 35(3):24–35. https://doi.org/10.1109/MS.2018.2141039
15. Kecskemeti G, Marosi AC, Kertesz A (2016) The ENTICE approach to decompose monolithic services into microservices. In: 2016 international conference on high performance computing simulation (HPCS), pp 591–596 . https://doi.org/10.1109/HPCSim.2016.7568389
16. Khadka R, Reijnders G, Saeidi A, Jansen S, Hage J (2011) A method engineering based legacy to soa migration method. In: Software maintenance (ICSM), 2011 27th IEEE international conference on, pp 163–172. IEEE
17. Lewis G, Smith D, Kontogiannis K (2010) A research agenda for service-oriented architecture (soa): Maintenance and evolution of service-oriented systems. Technical report. CMU/SEI-2010-TN-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA . http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9285
18. de Lima Justino Y, da Silva CE (2018) Reengineering legacy systems for supporting SOA: a case study on the Brazilian's secretary of state for taxation. In: Proceedings of the 40th international conference on software engineering: companion proceedings, ICSE '18, pp 125–126. ACM, New York, NY, USA . https://doi.org/10.1145/3183440.3195067
19. Lin J, Lin LC, Huang S (2016) Migrating web applications to clouds with microservice architectures. In: 2016 International conference on applied system innovation (ICASI), pp 1–4 . https://doi.org/10.1109/ICASI.2016.7539733
20. Millett S (2015) Patterns, principles and practices of domain-driven design. Wiley
21. Newman S (2015) Building microservices, 1st edn. O'Reilly Media, Inc
22. Newman S (2019) Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media, Incorporated . https://books.google.com.br/books?id=iul3wQEACAAJ
23. Pressman RS, Maxim BA (2016) Software engineering: a practitioner's approach, 8[a] Edition. McGraw Hill
24. Reddy VK, Dubey A, Lakshmanan S, Sukumaran S, Sisodia R (2009) Evaluating legacy assets in the context of migration to soa. Softw Quality J 17(1):51–63. https://doi.org/10.1007/s11219-008-9055-6
25. Richards M (2015) Microservices versus service-oriented architecture. O'Reilly Media
26. Santika H, Suhardi, Yustianto P (2017) Engineering local government financial service system under good governanceprinciples: case study: Cimahi government city. In: 2017 5th international conference on information and communication technology (ICoIC7), pp 1–6 . https://doi.org/10.1109/ICoICT.2017.8074657
27. Sheikh MAA, Aboalsamh HA, Albarrak A (2011) Migration of legacy applications and services to service-oriented architecture (soa). In: The 2011 international conference and workshop on current trends in information technology (CTIT 11), pp 137–142 . https://doi.org/10.1109/CTIT.2011.6107949
28. Vernon V (2013) Implementing domain-driven design. Pearson Education, Inc
29. Wagner C (2014) Model-driven software migration: a methodology reengineering, recovery and modernization of legacy systems. Springer. https://doi.org/10.1007/978-3-658-05270-6
30. Waseem M, Liang P, Shahin M (2020) A systematic mapping study on microservices architecture in DevOps. J Syst Softw 170. https://doi.org/10.1016/j.jss.2020.110798
31. Zimmermann O (2017) Microservices tenets. Comput Sci Res Develop 32(3):301–310. https://doi.org/10.1007/s00450-016-0337-0