# Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems

**Alessandra Levcovitz[1], Ricardo Terra[2], Marco Tulio Valente[1]**

[1]Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil

[2]Universidade Federal de Lavras (UFLA), Lavras, Brazil

`alessandralev@gmail.com,terra@dcc.ufla.br,mtov@dcc.ufmg.br`

*Abstract. The idea behind microservices architecture is to develop a single large, complex, application as a suite of small, cohesive, independent services. On the other way, monolithic systems get larger over the time, deviating from the intended architecture, and becoming tough, risky, and expensive to evolve. This paper describes a technique to identify and define microservices on a monolithic enterprise system. As the major contribution, our evaluation demonstrate that our approach could identify good candidates to become microservices on a 750 KLOC banking system, which reduced the size of the original system and took the benefits of microservices architecture, such as services being developed and deployed independently, and technology independence.*

## 1. Introduction

Monolithic systems inevitably get larger over the time, deviating from their intended architecture and becoming hard, risky, and expensive to evolve [6, 8]. Despite these problems, enterprise systems often adopt monolithic architectural styles. Therefore, a major challenge nowadays on enterprise software development is to evolve monolithic system on tight business schedules, target budget, but keeping quality, availability, and reliability [3].

Recently, microservices architecture emerged as an alternative to evolve monolithic applications [2]. The architecture proposes to develop a system as a set of cohesive services that evolve over time and can be independently developed and deployed. Microservices are organized as a suite of small services where each one runs in its own process and communicates through lightweight mechanisms. These services are built around business capabilities and are independently deployed [5]. Thereupon, each service can be developed in the programming language that suits better the service characteristics, can use the most appropriate data persistence mechanism, can run on an appropriate hardware and software environment, and can be developed by different teams. There are many recent success stories on using microservices on well-known companies, such as Amazon and Netflix [7].

Therefore, microservices are an interesting approach to incrementally evolve enterprise application. More specifically, new business functions can be developed as microservices instead of creating new modules on the monolithic codebase. Moreover, existing components can be extracted from the monolithic system as microservices. This process may contribute to reduce the size of monolithic application, and create smaller and easier to maintain code.

In this paper, we describe a technique to identify microservices on monolithic systems. We successfully applied this technique on a 750 KLOC real-world monolithic banking system, which manages transactions from 3.5 million banking accounts and performs nearly 2 million authorizations per day. Our evaluation shows that the proposed technique is able to identify microservices on monolithic system and that microservices can be a promising alternative to modernize legacy enterprise monolithic system.

The remainder of this paper is organized as follows. Section 2 describes the proposed technique to identify microservices on monolithic systems. Section 3 evaluates the proposed technique on a real-world monolithic banking system. Section 4 presents related work and Section 5 concludes the paper.

## 2. The Proposed Technique

The proposed technique considers that monolithic enterprise applications have three main parts [6]: a *client side* user interface, a *server side* application, and a *database*, as shown in Figure 1. It also considers that a large system is structured on smaller subsystems and each subsystem has a well-defined set of business responsibilities [1]. We also assume that each subsystem has a separate data store.
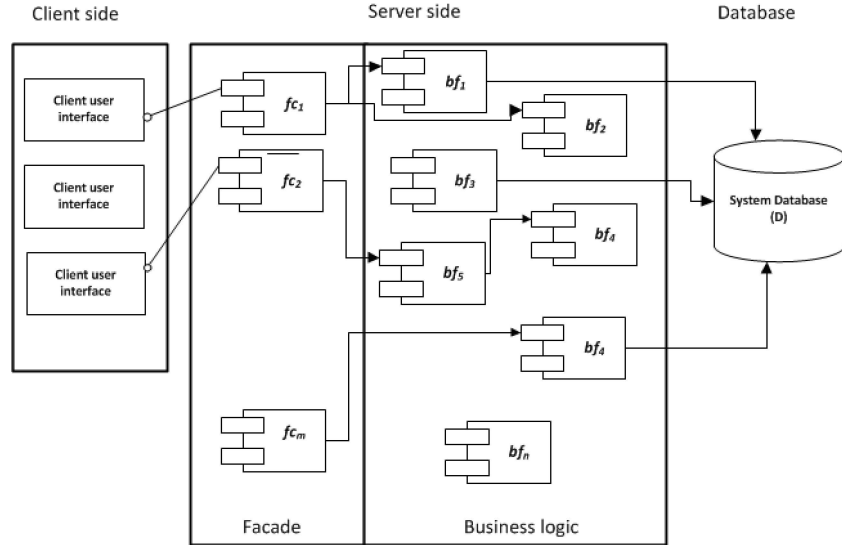


**Figure 1. Monolithic application**

In formal terms, we assume that a system $S$ is represented by a triple $(F, B, D)$, where $F = \{fc_1, fc_2, \ldots, fc_{n'}\}$ is a set of facades, $B = \{bf_1, bf_2, \ldots, bf_{n''}\}$ is a set of business functions, and $D = \{tb_1, tb_2, \ldots, tb_{n'''}\}$ is a set of database tables. Facades ($fc_i$) are the entry points of the system that call business functions ($bf_i$). Business functions are methods that encode business rules and depend on database tables ($tb_i$). It is also important to define an enterprise organization $O = \{a_1, a_2, \ldots, a_w\}$ is divided into business areas $a_i$, each responsible for a business process.

We describe our technique to identify microservices on a system $S$ in the following steps:

**Step #1**: Map the database tables $tb_i \in D$ into subsystems $ss_i \in SS$. Each subsystem represents a business area ($a_i$) of organization $O$. For instance, as presented in Figure 2, subsystem $SS_2$, which represents business area $a_2$, depends on database tables $tb_3$ and $tb_6$. Tables unrelated to business process—e.g., error messages and log tables—are classified on a special subsystem called *Control Subsystem* (SSC).
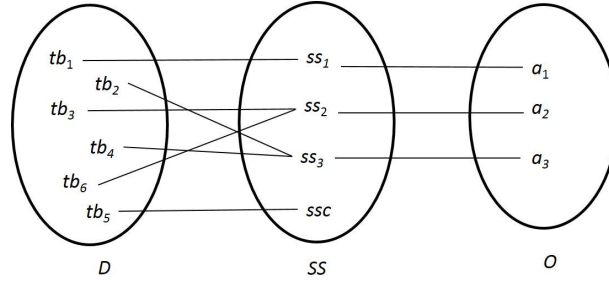


**Figure 2. Database decomposition**

**Step #2**: Create a dependency graph $(V, E)$ where vertices represent facades ($fc_i \in F$), business functions ($bf_i \in B$), or database tables ($tb_i \in D$), and edges represent: (i) calls from facades to business functions; (ii) calls between business functions; and (iii) accesses from business functions to database tables. Figure 3 illustrates a graph where facade $fc_2$ calls business function $bf_2$ (case *i*), business function $bf_2$ calls business function $bf_4$ (case *ii*), and business function $bf_4$ accesses database tables $tb_2$ and $tb_3$ (case *iii*).
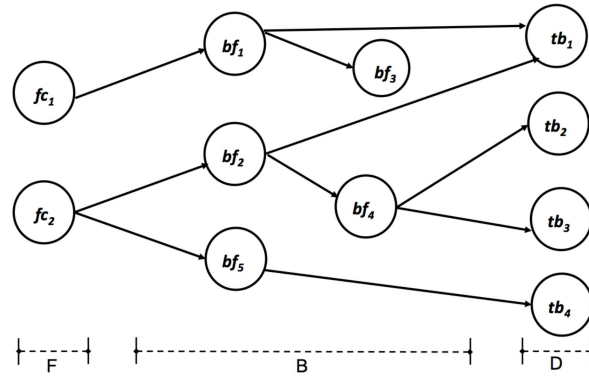


**Figure 3. Dependency Graph**

**Step #3**: Identify pairs $(fc_i, tb_j)$ where $fc_i \in F$ and $tb_j \in D$, and there is a path from $fc_i$ to $tb_j$ on the dependency graph. For instance, in the graph illustrated in Figure 3, we identify the pairs $(fc_1, tb_1), (fc_2, tb_1), (fc_2, tb_2), (fc_2, tb_3),$ and $(fc_2, tb_4)$.

**Step #4**: For each subsystem $ss_i$ previously defined in Step #1, select pairs $(fc_i, tb_j)$ identified on the prior step where $tb_j \in ss_i$. For instance, in our illustrative example, $ss_3 = \{tb_2, tb_4\}$ then we select the pairs $(fc_2, tb_2)$ and $(fc_2, tb_4)$.

**Step #5**: Identify candidates to be transformed on microservices. For each distinct pair $(fc_i, tb_j)$ obtained on the prior step, we inspect the code of the facade and business func-

tions that are on the path from vertex $fc_i$ to $tb_i$ in the dependency graph. For instance, for pair $(fc_2, tb_2)$, we inspect facade $fc_2$ and business functions $bf_2$ and $bf_4$. The inspection aims to identify which business rules actually depend on database table $tb_j$ and such operations should be described in textual form as rules. Therefore, for each pair $(fc_i, tb_j)$, a candidate microservice (M) is defined as follows:

- *Name*: the service name according to pattern [subsystemname].[processname]. For instance, Order.TrackOrderStatus.

- *Purpose*: one sentence that describes the main business purpose of the operation, which is directly associated to the accessed database entity domain. For instance, *track the status of a customer order*.

- *Input/Output*: the data the microservice requires as input and produces as output—when applied—or expected results for the operation. For instance, microservice Order.TrackOrderStatus requires as input the *Order Id* and produces as output a *List of finished steps with conclusion date/time* and a *List of pending steps with expected conclusion date*.

- *Features*: the business rules the microservice implements, which are described as a verb (action), an object (related to the database table), and a complement. For instance, *Identify the order number*, *Get the steps for delivery for type of order*, *Obtain finished steps with date/time*, and *Estimate the date for pending steps*.

- *Data*: the database tables the microservice relies on.

**Step #6**: Create API gateways to turn the migration to microservices transparent to clients. API gateway consists of an intermediate layer between client side and server side application. It is a new component that handles requests from client side—in the same technology and interface as $fc_i$—and synchronizes calls to the new microservice M and to $fc_i'$—a new version of $fc_i$ without the code that was extracted and implemented on microservice M. An API gateway should be defined for each facade.

There are three cases of synchronization we have to consider in our evaluation: (i) when the input of $fc_i'$ is the output of M or the input of M is the output of $fc_i'$; (ii) when the input of M and $fc_i'$ are the same as API gateway input and the instantiation order is irrelevant; and (iii) when we have to split $fc_i$ into two functions $fc_i'$ and $fc_i''$ and microservice M must be called after $fc_i'$ and before $fc_i''$.

On one hand, if we can synchronize the calls as described on case (i) or (ii), we identify the proposed microservice M as a *"strong candidate"*. On the other hand, if we can only synchronize the calls as in case (iii), we identify the proposed microservice as a *"candidate with additional effort"*. Particularly in our technique, assuming a microservice of a subsystem $ss_x$, if we identify a business rule in the microservice definition that needs to update data in $tb_i \in ss_x$ and $tb_j \notin ss_x$ in the same transaction scope, we identify the proposed microservice as a *"non candidate"*.

When every evaluated pair $(fc_i, tb_j)$ of a subsystem is classified as microservice candidate, we recommend to migrate the entire subsystem to the new architecture. In this case, we have to implement the identified microservices, create an independent database with subsystem tables, develop API gateways, and eliminate the subsystem implementation (source code and tables) from system S. Although API gateways must be deployed

in the same server as system S to avoid impacts on client side layer, the microservices can be developed using any technology and deployed wherever is more suitable.

## 3. Evaluation

We applied our proposed technique on a large system from a Brazilian bank. The system handles transactions performed by clients on multiple banking channels (Internet Banking, Call Center, ATMs, POS, etc.). It has 750 KLOC in the C language and runs on Linux multicore servers. The system relies on a DBMS with 198 tables that performs, on average, 2 million transactions a day.

**Step #1**: We identified 24 subsystems including subsystem SSC. Table 1 shows a fragment of the result obtained after this initial step. Headers represents subsystems and their content represent the tables they rely on. One problem we identified is that certain tables—highlighted in grey—are associated to more than one subsystem.

#### Table 1. Mapping of Subsystems and Tables

| Business Actions | Service Charges | Checks | Clients | Current Accounts | Saving Accounts | Social Bennefits | Pre-approved credit | Debit and Credit cards | SMS Channel |
|---|---|---|---|---|---|---|---|---|---|
| ACO | AGT | CHS | CLT | *CNT* | *CNT* | BEN | LPA | CMG | CTS |
| ACB | ISE | TCE | | CCT | CPO | DPB | | INP | STS |
| RCA | PTC | ECH | | *RCC* | *LAN* | DBC | | NPP | CMS |
| | PTF | HET | | *LAN* | *RCC* | IBS | | BIN | RCS |
| | RTE | CCF | | CCO | MPO | LBE | | CCM | RLS |
| | RTT | | | CCE | PPO | | | PBE | RTS |
| | TPT | | | CHE | SPA | | | | |
| | TTE | | | | | | | | |
| | UTM | | | | | | | | |
| | DUT | | | | | | | | |

**Step #2**: We created a dependency graph composed by 1,942 vertices (613 facades, 1,131 business functions, and 198 database tables), 5,178 edges representing function calls and 2,030 edges corresponding to database table accesses. Due to the size of our evaluated system, we proceed our evaluation to the following five subsystems: *Business Actions*, *Service Charges*, *Checks*, *Clients* and *SMS channel*. Table 2 illustrates the characteristics of each subsystem. For subsystem *Business Actions*, we obtained the graph presented on Figure 4.

#### Table 2. Evaluated Subsystems

| Subsystem | Business actions | Service Charges | Checks | SMS channel | Client |
|---|---|---|---|---|---|
| Tables (vertices) | 3 | 10 | 5 | 6 | 1 |
| Functions (vertices) | 5 | 62 | 29 | 138 | >150 |
| Function calls (edges) | 3 | 79 | 14 | 133 | >150 |
| Database accesses (edges) | 6 | 14 | 22 | 140 | 26 |
| Microservices candidates | 1 | 3 | 8 | 4 | 4 |

**Steps #3–#4**: Considering subsystem *Business Actions*, we find the following pairs:

$$(\texttt{AUTCCErspSolAutCceNov}, \texttt{ACO}), \qquad (\texttt{AUTCCErspSolAutCceNov}, \texttt{RCA}),$$
$$(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACO}), \qquad (\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{RCA}),$$
$$(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACB}).$$
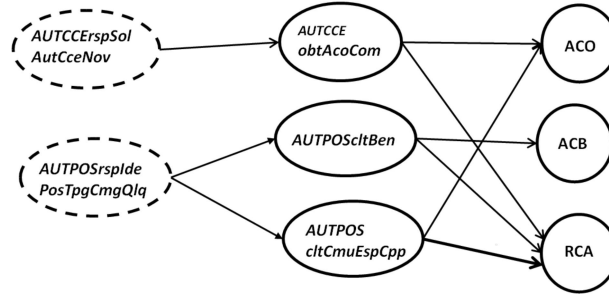


**Figure 4. Business Actions Subsystem Graph**

**Step #5**: For pairs $(\texttt{AUTCCErspSolAutCceNov}, \texttt{ACO}), (\texttt{AUTCCErspSolAutCceNov}, \texttt{RCA})$ obtained on prior step, we inspect the code of facade `AUTCCErspSolAutCceNov` and the business functions it calls (`AUTCCEobtAcoCom`), and we identified the microservice described as follows:

- *Name*: `BusinessActions.ListBusinessActionsForAccount`.

- *Purpose*: List business actions related to an account on the relationship channel.

- *Input/Output*: Account number and channel id as input, and a list of business actions as output.

- *Features*: Retrieve business actions assigned to the account; retrieve business actions enabled for the relationship channel; and retrieve the list of business actions assigned to the account and enabled for the channel.

- *Data:* Database tables ACO and RCA.

We also evaluated the other three pairs $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACO})$, $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{RCA})$, $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACB})$ and the business functions they call `AUTPOScltBen` and `AUTPOScltCmuEspCpp`, and we identified the same microservices as the one described above. In fact, table ACB could be merged with ACO.

**Step #6**: For facades `AUTCCErspSolAutCceNov` and `AUTPOSrspIdePosTpgCmgQlq`, we identified an API gateway that suits case ($i$) described in the proposed technique.

We also evaluated steps #3 to #6 for subsystems: *Service Charges, Checks, Clients* and *SMS channel*. Although subsystem *Service Charge* has 10 tables and 51 facades, we only identified and defined the following three microservices: `ServiceCharge.CalculateServiceCharge`, `ServiceCharge.IncrementServiceChargeUsage`, and `ServiceCharge.DecrementServiceChargeUsage`. On one hand, we identified 14 API gateways that also suit case ($i$). On the other hand, we identified other 37 API

that suit case (*ii*). However, we can avoid the development of the last 37 API gateways since the called microservices have only input data and can be implemented with an asynchronous request. Thus, particularly in this case, we suggest to use a message queue manager (MQM) for communication and substitute the C code that performs an update on database table for a "put" operation on a queue.

For subsystems *Checks* and *SMS channel*, we identified microservices and APIs with the same characteristics of subsystem *Service Charge*, which indicates that both subsystems are good candidates to be migrated to microservices. Nevertheless, for subsystem *Client*—which accesses only one table—we identified one microservice that must be called by more than 50 API gateways that suits case (*iii*). Therefore, we did not recommend its migration to microservices, since the effort to split and remodularize more than 50 functions, create and maintain more than 50 API gateway are probably greater than the benefits of microservice implementation.

Last but not least, we disregard subsystems that have one or more tables that appear in more than one subsystem list, such as table CNT of subsystem *Current Account* (refer to Table 1) because our technique to identify microservices considers that only one subsystem handles operation on each table.

**Brief discussion:** We classify our study as well-succeeded because we could identify and classify all subsystems, and create and analyze the dependency graph that helped considerably to identify microservices candidates. As our practical result, we recommended to migrate 4 out of the 5 evaluated subsystems to a microservice architecture.

## 4. Related Work

Sarkar et al. described a modularization approach adopted on a monolithic banking system but they did not use a microservice architectural approach [8]. Richardson evaluated a microservice architecture as a solution for decomposing monolithic applications. His approach considered the decomposition of a monolithic system into subsystems by using use cases or user interface actions [6]. Although this is an interesting approach, in some situations a use case represents a set of operations of different business subsystems which are synchronized by an actor action. Therefore, we do not always have an entirely system described with use cases. By contrast, our technique starts by evaluating and classifying the database tables into business subsystems, which demands access only to the source code and the database model. Namiot et al. presented an overview of microservices architecture but they do not evaluate the concepts on a real-world system [4]. Terra et al. proposed a technique for extracting modules from monolithic software architectures [9]. This technique is based on a series of refactorings and aims to modularize concerns through the isolation of their code fragments. Therefore, they do not target the extraction of modules that can be deployed independently of each other, which is a distinguish feature of microservice-based architectures.

## 5. Conclusion and Future Work

This paper describes a technique to identify microservices on monolithic systems. We successfully applied the proposed technique on a 750 KLOC real-world monolithic banking system, which demonstrated the feasibility of our technique to identify upper-class

microservices candidates on a monolithic system. The graph obtained for each subsystem helped considerably to evaluate the functions, identify, and describe microservices.

There are subsystems that were not classified as good candidates to migrate to microservices, however. We found scenarios that would require a considerable additional effort to migrate the subsystem to a set of microservices. For instance, (i) subsystems that share same database table, (ii) microservice that represents an operation that is always in the middle of another operation, and (iii) business operations that involve more than one business subsystem on a transaction scope (e.g., money transfer from a check account to a saving account).

More important, the migration to the microservice architecture can be done incrementally. In other words, we can profit from the microservices architecture—e.g., services being developed and deployed independently, and technology independence— without migrating the entire system to microservices. Both kinds of systems architecture—monolithic and microservices—can coexist in a system solution. In fact, one challenge in using microservices is deciding when it makes sense to use it, which is exactly the ultimate goal of the technique proposed in this paper.

As future work, we plan to evaluate other subsystems of the banking system, implement the identified microservices, and measure in the field the real benefits. We also plan to evaluate the proposed technique on Java monolithic systems.

## Acknowledgment

## References

[1] Melvin E Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[2] Martin Fowler and James Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014.

[3] M Lynne Markus and Cornelis Tanis. The enterprise systems experience-from adoption to success. *Framing the domains of IT research: Glimpsing the future through the past*, 173:207–173, 2000.

[4] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[5] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[6] Chris Richardson. Microservices: Decomposing applications for deployability and scalability. http://www.infoq.com/articles/microservices-intro, 2014.

[7] Chris Richardson. Pattern: Microservices architecture. http://microservices.io/patterns/microservices.html, 2014.

[8] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35, 2009.

[9] Ricardo Terra, Marco Tulio Valente, and Roberto S. Bigonha. An approach for extracting modules from monolithic software architectures. *IX Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–18, 2012.