International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.18, April 2020

Mind Overflow: A Process Proposal for Decomposing Monolithic Applications in Microservices

Tcharles Pereira da Silva
Applied computing graduate program (PPGCA),
University of Vale do Rio dos Sinos,
São Leopoldo, RS, Brazil

Kleinner Farias
Applied computing graduate program (PPGCA),
University of Vale do Rio dos Sinos,
São Leopoldo, RS, Brazil

ABSTRACT

Constant changes made by different developer profiles turn legacy applications into monolithic ones. Although it is a known issue, little has been done to mitigate it. This paper proposes Mind Overflow, a process to guide the decomposition of a monolithic application to a microservice architecture. With Mind Overflow, researchers and developers benefit from the use of consolidated design patterns, architectures, and technologies through a comprehensive decomposition workflow. The case study showed promising results, indicating that Mind Overflow is feasible to break down monolithic to a microservice-based architecture, including reducing cyclomatic complexity and producing highly cohesive microservices.

General Terms

Software system, Software maintenance

Keywords

Decomposition process, software architecture

1. INTRODUCTION

Modern software systems or legacy applications undergo various changes throughout their lifecycle, which are usually performed by different developers [17]. These legacy applications are built and maintained in increasingly fast-changing business environments in industry [20]. Usually, unstable requirements and rapidly advancing information technologies are ever-present characteristics of projects in real-world settings. These constant changes made by different developer profiles turn legacy applications into monolithic ones. Typically, these changes impact on various features of systems under maintenance [3], and developers often have to compare different versions of system design created [12]. Empirical studies demonstrate that these integrations and maintenance are time-consuming and error prone [5][8][9][10].
Although several techniques help monolithic application code to be error-free [2][22], this is not synonymous with easy-to-maintain applications. Typically, by satisfying customers' business needs without error, applications are delivered, in turn completing the development cycle.
An application considered legacy in a company, usually suffers from technical managers and maintainers, because it was developed using techniques that are not being used by the current developers of the company; however, such a solution is still vital and valuable to the business of those who use it [1]. While this issue of maintaining legacy applications is well known, little has been done to mitigate it. The use of microservice-based architectures has shown promise in this regard. Microservice can be defined as an application composed of small features (or services) that take into consideration the business context, that is autonomous and independent in its execution and publication, doing this in an automated way and that can communicate with each other, besides providing endpoints to the external environment of its limits [11].
This work, therefore, proposes Mind Overflow, a proposal of technology and framework agnostic process, which aims to guide the decomposition of a monolithic application to a microservices architecture through a systematic sequence of small decompositions. Using Mind Overflow, researchers and developers benefit from the use of well-matched design patterns, architectures, and technologies through an understandable and consistent workflow. Mind Overflow differentials are as follows: (1) definition of a software re-engineering process based on iterations that can evolve incrementally; and (2) introduction of a workflow of decomposition of monolithic architectures, seeking to modulate the main responsibilities of the microservices application. The case study showed promising results, indicating that Mind Overflow is feasible to break down monolithic to microservice-based architectures, including reducing cyclomatic complexity and producing highly cohesive microservices.
This paper is organized as follows: Section 2 presents the fundamental concepts for understanding the work. Section 3 covers related work. Section 4 presents the proposed work. Section 5 describes the evaluation and how the results were analyzed. Section 6 discusses the conclusions and future work.

2. BACKGROUND

This section aims to outline the main concepts required to understand the proposed process.

2.1 Software Architecture

The styles and architectural patterns used in systems should be driven by the actual demands and specific qualities, meeting and exceeding the expectations of their use [22]. Avoiding excessive use of patterns and styles becomes as fundamental as applying the architecture correctly. A good basis to guide the choice of these

styles and patterns is to justify their purpose for the solution based on functional requirements, otherwise, they should be eliminated from the system.

Architectural layers. This architectural standard supports N layers and is commonly used in web, enterprise, and desktop applications. The basic mode of use of this standard is three layers: user interface, application layer, and infrastructure layer. Being a good practice premise of this pattern, a layer should only interact with its layer or layers below it, thus avoiding calls to higher layers [22]. The proposed process in this work has the premise that the generated systems use a layered architecture.

Service Oriented Architecture. A proposal for the implementation of SOA is the use of a hexagonal architecture, which has adapters for data entry, which can be REST, SOAP and messaging through messaging services. With entry points that can be distinct, they all direct requests to a common point that would be the application layer, which subsequently directs them to the domain model for business and purpose-based processing for that request [22].

Event Oriented Architecture. According to Vernon [22], it is a standard capable of promoting the production, detection, consumption, and reaction to events. These events are not intended to simply be decorative technical notifications, but rather represent the occurrences of business process activities that will be routed to all reception points of these activities, which will normally perform other activities arising from the identified initial activity. To receive this notification, whether it is synchronous or asynchronous, the producer and consumer standard are used, allowing different applications to communicate with each other from the chain of events. The process developed in this paper uses the producer and consumer standard for communication between microservices, where each application acts with a different domain responsibility.

2.2 Domain-Driven Design

The Domain-Driven Design (DDD) software development approach, created by Eric Evans [7], comes to assist in the way software expresses the solution for the business, so that the models generated explicitly demonstrate the intended purpose of the business. DDD puts domain experts and developers on a level playing field as a cohesive and united team, making the software that is developed makes sense for the business, not just the coders, with centralized business knowledge source code and no longer in select groups of people, which are usually the developers.

Ubiquitous language. According to Evans [7], one of the biggest problems in building software is the poor understanding of business goals motivated by improper communication between business experts and developers. For domain experts, technical implementation and design terms are not easily understood, as their view of the business and how it works often uses domain-specific jargon. Developers who master the technical terms and design terms have a hard time understanding the objectives and understanding of what is being addressed by the experts, as a system is usually discussed and seen only at the implementation level by developers, without talking about the huge amount of translations applied to the language of the experts turn the language of the developer. Developers who work in various parts of the business usually create their concepts about the meaning of terms, often making no sense to the business itself.

Bounded context. It limits the scope of understanding and implementation of a particular model to its participation with the business. Each delimited context is defined as a ubiquitous language that should be known to all team members [7]. All responsibilities of the model must be well defined and its understanding of the other contexts must be defined, although knowledge of translations and integrations between the delimited contexts is somewhat separate. For the implementations of this work, a microservice will always be treated in bounded context form.

Context map. According to Vernon [22], a context map defines how bounded contexts will communicate with each other, giving a global context about the business solution. Relationships between the contexts delimited in the context map can be classified and translated according to the alignment between the teams. For this work, the context map will be generated from Event Storming, a technique created by Brandoline [4], which is defined as the first part of the proposed process, having the communications identified as event responses following of business activities.

Domain events. The domain event is defined by publishing activities in the form of notifications by publishers to their subscribers, which may be from the application itself or from other applications that are subscribed to receive notifications [22]. The process proposal of this work considers domain events as part of the stage and its understanding is fundamental for the elaboration of user stories from Event Storming.

3. RELATED WORK

This section provides a comparative analysis of related work. The objective of the analysis is to enrich the process that will be elaborated based on the studies already done on the decomposition of monolithic applications in microservices.

3.1 Selection criteria

This work used Google Scholar as a database, researching the theme of decomposition of monolithic systems for a microservice architecture through the keywords “microservice” and “monolith”. After the first survey on the subject of decomposition for microservices, only works that aimed to approach or construct or restructure applications for microservices using some kind of technique, process or methodology were filtered. A second filter used was the consideration of articles published from 2015 to 2019, due to microservices being a very recent theme.

3.2 Work analysis

This performs a comparative analysis of five works that explore the theme of decomposition of monolithic systems for microservices, either by technique, process or methodology, in order to identify what are the common criteria between the works.

Levcovitz et al. [16] proposed a technique for the decomposition of monolithic systems into small cohesive and independent services. The technique was applied to a banking system that had about 750,000 lines of source code, 2 million bank accounts, operating approximately 2 million daily authorizations. As a premise for the work, the flow of a simple iteration into the decomposed system is part of the user interface, reaching a facade that is responsible for assigning the request to the application business functions. From the business function, one or more business functions can be called chained and can use one or more database tables, each business function being classified in an area, which the article treats as an organizational unit. The technique consists of six steps: the first step is to identify subsystems according to business areas and database tables; The second step creates a graph of dependencies between facades, one or more business functions, and database tables. In the third step, based on the generated dependency graph, the links between facade and table of the database are mapped; In the fourth step, for each subsystem identified in the first step, the facades that

are related to the subsystem database tables are mapped; The fifth step is responsible for identifying and evaluating candidates to follow in the decomposition. In the last step, an API Gateway is created that acts as an orchestrator of requests for microservices, enabling client access to the server, in order to ensure transparency for users.

Knoche & Hasselbring [15] present an application modernization technique based on the decomposition of legacy systems for a microservices architecture. The legacy system used for the foundation and application of the technique is a client management system developed in Cobol with over one million lines of code, started in the 1970s and 1980s. The proposed modernization technique is divided into five steps, which are: (1) the first step is to identify the domain services that will provide the functionality expected by customers, after that, an analysis is made to identify the application entry points, which are any form of access used by other applications; (2) the second step is to implement the facades in the form of adapters to receive the requests, always considering that these implementations must guarantee the same behavior of the operations already performed by the system, and to have this guarantee, the team used software testing techniques, generating reliability in the expected processing in the implementations; (3) for the third step, the client applications are migrated to the newly created facades. Thus, no longer accessing the legacy application directly, limiting entry points through facades; (4) the fourth step establishes the internal facades of the application, in order to organize the functionality requests for internal facades, and no longer for other functionalities.

Gysel et al. [13] propose a tool for decomposition of monolithic applications in microservices, based on the experiences of various software architects. With a catalog containing 16 decomposition criteria, classified into four categories, the tool works on a user-supplied input in JSON format, which can be: a relationship entity model based on the pattern database, entities, and aggregates modeling Domain-Driven Design or a use case template. The four categories for the criteria are: cohesion, compatibility, constraints and communication. Each criterion is formatted with a template very similar to that used in agile practices, containing information such as: identification and name; description; specification artifacts in the system; literature references, category and characteristics.

Gysel et al. [13] propose a tool for the decomposition of monolithic applications in microservices, based on the experiences of various software architects. With a catalog containing 16 decomposition criteria, classified into four categories, the tool works on user-supplied input in JSON format, which can be: a relationship entity model based on the pattern database, entities, and aggregates modeling Domain-Driven Design or a use case template. The four categories for the criteria are cohesion, compatibility, constraints, and communication. Each criterion is formatted with a template very similar to that used in agile practices, containing information such as identification and name; description; specification artifacts in the system; literature references, category, and characteristics.

Escobar et al. [6] propose a process that helps the developer to decompose a monolithic application in microservices, through diagrams resulting from data layer analysis belonging to each Enterprise Java Beans (EJB) through clustering. The process is divided into three steps: the first layer called data injection, which deals with the processing of application source code, resulting in a KDM metamodel that is used as input for the next step; For the second stage, the classes, interfaces, and methods present in the metamodel generated by the first stage are identified, which are classified into two possible clusters that are EJB cluster and microservice cluster.

3.3 Comparative analysis of the works

This section will compare the works presented in section 3.3. Some criteria were elaborated for comparison purposes: main contribution; evaluation method; evaluation context; type of technique; microservice granularity and tool.

Table 1 provides an overview of the interplay between related work and comparative criteria. Regarding the main contributions, only the work of Gysel [13] proposed a tool for the decomposition of monolithic systems in microservices. The works of Levcovitz [16], Knoche [15] and Escobar [6] contribute to the creation of a process to perform the decomposition. The work of Rocha [21] proposes the use of an algorithm to perform the decomposition.

For the evaluation methods of the selected papers, all papers presented a case study of their proposals. In [13], in addition to the case study presented, a survey was also performed to ascertain the quality resulting from the proposed tool. The work environments of [13] and [6] were academia and industry, while the works of Levcovitz [16], Knoche [15] and Rocha [21] were applied exclusively in the industry. Of the selected articles, only the works of [13] and [6] used clustering forms in the development of the tool. Only the works by Rocha [21] and [6] allow configuration of the granularity in which microservices will be generated. Only the work of [13] presented a system for the decomposition of monolithic systems for microservices, whereas [6] brought a prototype.

3.4 Research opportunities

Identified through the analysis of the articles in section 3.3, the research opportunities raised by this paper are: (1) the creation of more processes and algorithms that guide software developers and architects in the microservices decomposition stage; (2) the application of decomposition independent of the contribution of work so that it can be evaluated by academia and industry, thus having parity of results that can be evaluated; (3) prototyping of model applications, representing coding standards, architectures used and technologies that contribute to the development and evaluation of results. From these opportunities, this work develops a process as a guide for the researcher and developer in decomposing their target application, the evaluation will be through a case study described in section 5.1.

4. PROPOSED APPROACH

This section introduces Mind Overflow, a technology-agnostic process, and a framework that aims to guide the decomposition of a monolithic application to a microservices architecture through a sequence of decomposition systematic.

4.1 Overview

Figure 1 presents an overview of the proposed approach, which consists of four steps: (1) Event Storming; (2) story storage and prioritization; (3) implementation of microservice; and (4) integration with the monolithic. This approach utilizes emerging techniques and methodologies, including Domain-Driven Design, Agile Methods, and Event Storming. Each step is described below.

Step 1: Perform Event Storming. This step focuses on gathering requirements and their specifications. The Event Storming technique was used for this purpose. Created by Brandoline [4], to engage the technical team and domain experts, the technique is characterized by the use of colored stickers in a chronologically and linearly organized discussion space, where the purpose is to understand of business processes. Usually, this storming activity is

Articles	Main contribution				Evaluation method			Evaluation context		Technique type	Granularity of microservices	Tool		
	Algorithm	Architecture	Tool	Process	Case study	Controlled Experiment	Research	Industry	Academy	Clustering	It is configurable	IDE Plugin	Prototype	System
Gysel et al.[13]	-	-	+	-	+	-	+	+	+	+	-	-	-	+
Levcovitz et al. [16]	-	-	-	+	+	-	-	+	-	-	-	-	-	-
Knoche & Hasselbring [15]	-	-	-	+	+	-	-	+	-	-	-	-	-	-
Rocha [21]	+	-	-	-	+	-	-	+	-	-	+	-	-	-
Escobar et al. [6]	-	-	-	+	+	-	-	+	+	+	+	-	+	-

Legend: Not supported (-), Supported (+)

Table 1. : An analysis of the related work based on comparative criteria

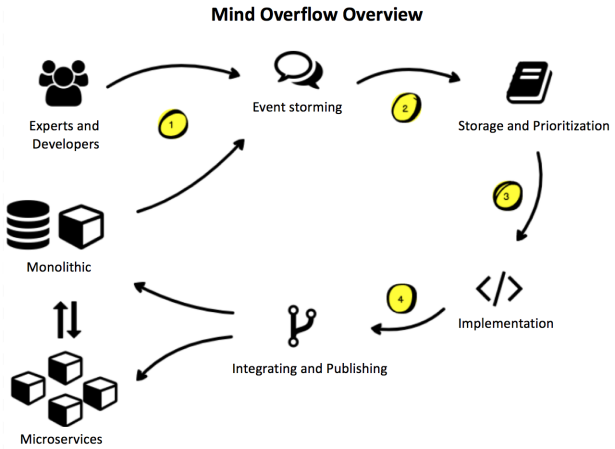

Fig. 1: An overview of the proposed approach.

divided not only in one day but in a few days so that what is being proposed is rethought by the participants, aiming at the maturity and effectiveness of the business ideas of the organization. For the proposed process, at this stage the following objectives must be achieved through Event Storming:

(1) Event Identification: are events that must be named in the past, when a product is created in the application, this event should be named as, for example, ProductCreated;

(2) Command Identification: it is the executions preceding the events that cause them to be triggered. Following the example, a command for the ProductCreated event could be the CreateProduct command;

(3) Microservice Identification: The goal of each microservice resulting from the application of this process is for each application domain, along with its sub-domains, to be present in one place, so that the language used in the domain is understood by all team members without ubiquity. Remembering that there may be communication between domains, this will be seen in the implementation step, when the event triggering part is explained;

(4) Story Creation: already with the defined events and the identified microservices, it is necessary to identify the behavior and flows that the business has. For each story, commands are executed and events are triggered for a purpose, and these steps are documented, it is the definition of a story.

Depending on the size and breadth of the monolithic application, hundreds of stories can be identified, which can make mapping the entire business unfeasible. The proposed process aims to be a facilitator. Thus, it is advisable to prioritize the core migration of the business in the first instance, along with the stories that are most relevant in the migration to microservices. Subsequently, the same process can be applied again for missing parts, without impact on what has already been worked on. Figure 2 describes the first step of Mind Overflow. With these objectives achieved, Step 2 will be performed.

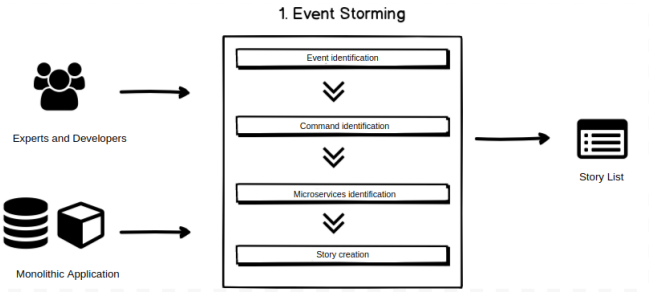

Fig. 2: Step 1: event storming

Step 2: Story Storage and Prioritization. Once stories are identified, they should be stored in a repository, also known as agile as Backlog. This repository can be from an excel spreadsheet to a task management application. The purpose of the Backlog is to store all pending tasks that will be performed in the next Sprints. Backlog tends to increase with the lifetime of the software. Thus, the activities belonging to him should always be prioritized by how much they will add to the business of the organization, also considering the cost for its development. If the organization does not have a method for classifying activities, this work suggests weighing these activities according to the perception of the team or the Product Owner. For this, the values 1, 2, 3, 5, 8, 13 and 21 can be considered, something similar to Planning Poker. Activities that have the highest added value should be prioritized. Prioritization of activities is not the process proposal, i.e., the organization is free to

classify the activities that will be considered for the next step. Figure 3 describes the second step of Mind Overflow. With the stories classified, described and defined as Sprint, Step 3 is performed.

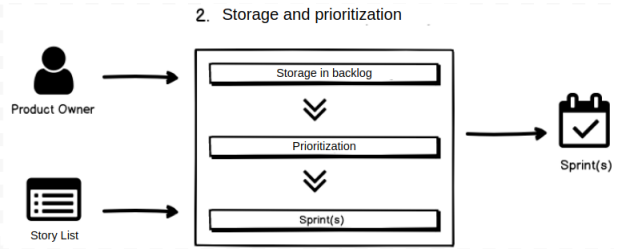

Fig. 3: Step 2: storage and prioritization

Step 3: Implementation. For the implementation stage, some assumptions must be made for the continuity of the microservice construction project. It is critical that the microservice team is aligned with the ubiquitous language used in the application domain, and that they have understood the purpose for which the microservice is being created. This can be achieved at planning meetings if the company follows the Scrum methodology. The solution design shall follow an MVC layered architecture model. Another highly recommended approach is for the microservice to be built to provide API's, that is, it has resources made available through HTTP requests, whether authorized requests or not, depending on the rules of each microservice. Another suggested feature is the execution of periodic commands programmed by the application itself, thus, the application is independent of the operating system configuration to execute its chrons, routines executed repeatedly over a certain time. Figure 4 shows the code structure of a feature developed based on Mind Overflow.

```
class UpdateProductFeature implements Feature
{
    private $slug;
    private $data;

    public function __construct($slug, $data)
    {
        $this->slug = $slug;
        $this->data = $data;
    }

    public function run()
    {
        $product = (new UpdateProductJob($this->slug, $this->data))->run();
        (new ProductUpdateEvent($product))->run();

        return $product;
    }
}
```

Fig. 4: Example feature implemented using Mind Overflow approach

With these assumptions satisfied, the development is divided into five phases shown in Figure 5 and described below:

(1) **Feature creation:** the idea is that each user story becomes a class-like feature for the microservice, keeping centrally and representing all story responsibilities through the execution of jobs and operations. Ideally, there is only one public method in the feature that will be responsible for calling the other features, so with a nomenclature that expresses the purpose of this feature, it is possible for any developer to easily understand what are the impacts on changing a particular feature. flow and what's contained in this small business approach, thus becoming a code that teaches the reader what the domain's characteristics are, without the need for deep debugging of functionality. A feature can also trigger events and be called through event reception and through controllers (which can be triggered through REST requests and periodic task crons);

(2) **Job creation:** The main purpose of the job is to be responsible for a restricted execution that has only one responsibility in the system flow. Here should be considered the principle of Single Responsibility, i.e., this job should have only one reason to exist and also the Open-Closed standard, which is when a class must be closed for change. This way the code is highly decoupled, making it possible to use mocks for unit tests, which should be created at job level. As a suggestion, a job should not call other jobs or invoke events for traceability and code standardization;

(3) **Operations creation:** so that there is no code duplication where features normally call the same jobs, the concept of operation is created. An operation should contain only calls to jobs, remembering that these calls, if changed, will impact other features that use this operation. As a suggestion, an operation should not fire events nor trigger other operations for the sake of traceability and code standardization;

(4) **Event shooting:** as this is an agnostic process to framework and programming language, it is up to the process user to choose a technology that has the possibility of triggering events through messaging services, such as RabbitMQ. The events triggered here are the same raised by Event Storming at the beginning of the process. The goal with the triggering of events by the application is that it is possible to partition and recognize underlying flows both by the application itself and in other applications that will be listening to the indication of a particular event, then take some systematic action, thus becoming a system. characterized reactive to events;

(5) **Event reception:** as in triggering events, the technology used by the implementers of the process must be able to communicate through messaging services such as RabbitMQ. When the microservice identifies an event to which it subscribes, the application should fire an implemented feature for this behavior. As described earlier, this new feature can launch a new event, thus forming a chain of executions that will satisfy some business feature.

Step 4: Integration with Monolithic. To integrate with the monolithic system incrementally and transparently, it will be necessary to develop an API Gateway, which will be responsible for receiving all monolithic application requests, and then directing to the given target microservice. For this first approach, the REST architecture will be used to perform requests by the monolithic system using the HTTP protocol. Once the API Gateway endpoints are defined, and these endpoints are directed to the already implemented microservices, the adaptation in the monolithic system is initiated so that the requests can be performed transparently, no longer depending on the legacy, but on the new implemented microservices. Figure 6 shows an example of API Gateway integration.

4.2 Main features and approach

The approach developed in this paper has the following characteristics: (1) be agnostic to the framework and programming language; (2) business visibility through source code; (3) be based on good software development practices, which are widely targeted today by organizations such as Clean Code [18] and SOLID [19]; (4) easy maintenance due to separation of responsibilities and rapid identification of execution flow; (5) naturally delivers an almost ready structure if the organization wishes to further minimize microservice, bringing it to the level of functionality, whether it be a feature,

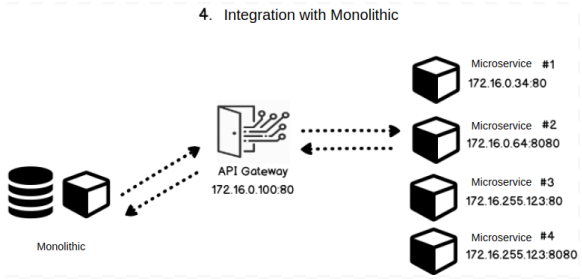

Fig. 6: Step 4: Integration with monolithic via API Gateway

a job or an operation; (6) horizontal scale viability of microservices using load balancer in infrastructure; (7) mastery of domain terms and language by all team members, as there are no business team translations for the development team; (8) allow the use of more than one programming language to better solve the business demand for microservices; (9) possibility of optimization and changes in the business process due to the rapid analysis from the impacts that they would generate; and, finally, (10) all jobs have unit tests, which are highly decoupled but cohesive to microservice.

4.3 Assumptions for use

For it to be successfully applied, some assumptions must be met: (1) use a language in the object-oriented paradigm for the creation of microservices; (2) possibility to change the monolithic application code to provide incremental and continuous advancement with the created microservices; (3) availability and accessibility to business specialists to assist in the definition of business processes; (4) definition of a ubiquitous language according to the microservices contexts and scopes, so that there are no translation from the business team to the development team; (5) developers with moderate knowledge of SOLID principles, particularly in relation to open-closed principle [19]; and (6) use or development of a framework for triggering and receiving events for communication between applications, preferably adopting some messaging software as a basis, for example, RabbitMQ.

5. EVALUATION

For the evaluation of this work, the case study methodology was used through the development of a monolithic application that was decomposed through the application of the Mind Overflow process.

5.1 Target application

For the case study, a monolithic e-commerce application created by the author was developed. Table 2 lists the features that are part of the monolithic application and will be broken down with the Mind Overflow process.
To obtain the desired decomposition model, the monolithic application was analyzed and, through good design and implementation practices, a new architecture involving microservices was implemented. This architecture, called "Desired Refactored Application," was compared to the result of applying the Mind Overflow process. That is, the desired refactored application will be compared to the refactored application using Mind Overflow. Figure 7 shows how modules were organized and the use of microservices for each application.

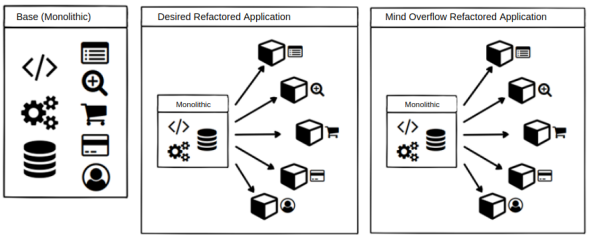

Fig. 7: Applications developed in this work

5.2 Technologies used

The technologies used for the three applications created were selected by the author based on the development experience gained after years of work in the field. Because the process approached as a focus of the study is agnostic to specific technologies, the tools mentioned here are not restrictive for process application and analysis:

— **Monolithic:** The monolithic application was developed using PHP 7.2, Laravel 5.8, Mysql 5.7, Docker 19.03.

— **Desired refactored application:** The desired refactored application was developed using PHP 7.2, Laravel 5.8 for the frontend application, Lumen 6.0 for the microservices, php-amqplib 2.10, Mysql 5.7, Docker 19.03 and RabbitMQ 3.7.

— **Application refactored through Mind Overflow:** The refactored application through Mind Overflow was developed using PHP 7.2, Laravel 5.8 for the frontend application, Lumen 6.0 for microservices, php-amqplib 2.10, Mysql 5.7, Docker 19.03 and RabbitMQ 3.7.

5.3 Metrics

Metrics used. Eight metrics were selected to allow assessment of different facets of applications, including lines of code, object-oriented programming (OOP) concepts, coupling, complexity, and bugs. Table 2 presents the selected metrics. These metrics were accounted for using the phpmetrics tool, which is an open-source project (https://phpmetrics.org). To perform the comparative analysis between the desired application and the generated application, distance calculations were used [14] between the desired application and the application generated by Mind Overflow. The distance represents the difference between the values presented by the metrics for the refactored application generated and refactored application through Mind Overflow.

5.4 Results

Table 3 presents the obtained results, considering two Catalog and Sales microservices. These microservices were chosen because they are representative to the others. Results will be displayed following the order of presentation of the metric groups.
Line of Code (LOC). Using the proposed approach helped to produce the desired functionality by achieving a distance of less than 50% in both microservices, as well as achieving the desired functionality. In the microservice Catalog, the distance represented 38.68% (294/760) of the desired code. In the microservice Sales, distance presented a higher value of 49.27% (643/1305). Code line removals can be done to get code as close as possible to the desired application. Mind Overflow was effective in terms of lines of code.
OOP. Through the proposal, the value of the class and method metrics increased concerning the desired application. Considering the

Group	Metric	Description
LOC	Lines of code	Measure the size of a computer program by counting the number of lines in the text of the program source code.
OOP	Classes	Number of classes in a computer program using object oriented programming.
	Methods	Number of methods of a computer program using object-oriented programming.
Coupling	Afferent coupling average	Represents the count of how many different classes refer to the current class by fields or parameters. If this number is high, this class has a high chance of being stable, reducing the risk of coupling.
	Efferent coupling average	Represents the count of how many different classes the current class references by fields or parameters. If a class has high efferent coupling, it means that it depends on many classes.
Complexity	Average cyclomatic complexity per class	Count of the number of independent paths that can be executed for each method. The result of cyclomatic complexity indicates how many tests need to be performed to verify all possible flows.
	Relative average system complexity	Metric composed of complexity within and between functions. It measures the complexity of system design in terms of function calls, parameter passing, and data usage.
Bugs	Average bug per class	Based on the number of operators (method names, arithmetic operators) and the number of operands. Halstead metrics give you an idea of how complex individual lines of code (or instructions) are. Halstead Bugs attempts to estimate the number of bugs that may be in a specific code snippet.

Table 2. : The metrics used in the evaluation.

metric number of classes, for the Catalog microservice, the distance represented 88.89% (24/27). In the microservice Sales, the distance obtained was 108.70% (50/46). In the metric number of methods, for the Catalog microservice, the distance represented 36.25% (29/80). For the micro-service of distance selling, it represented 50% (65/130). One possible reason why both metrics have increased is due to the use of the open and close responsibility principles for the creation of features and jobs, which are specific responsibility, creating a class for each responsibility, and consequently, method (s) for these classes. The benefit of using this standard will be seen in the complexity metrics.

Coupling. Both evaluated microservices obtained an increase in afferent and efferent couplings to the desired application. For the afferent coupling mean metric, in the Catalog microservice, the distance represented 120.83% (0.58/0.48). In the microservice Sales, the calculated distance was 113.43% (0.76/0.67). In the metric referring to the efferent coupling average, for the Catalog microservice, the distance represented 21.62% (0.4/1.85). For the microservice of sales, the distance represented 24.88% (0.53/2.13). Increasing the average value of the afferent coupling results in an increase in the chances of the class being stable, reducing coupling risks. The increase in efferent coupling means that a class depends on a larger number of classes, and for both microservices, this value is less than 50%, Mind Overflow is considered effective in terms of coupling.

Complexity. Through the Mind Overflow process, all assessed complexity metrics achieved their average reductions. For the mean cyclomatic complexity by class, in the Catalog microservice, the distance represented 13.89% (0.2/1.44). In the microservice of Sales, the calculated distance was 21.02% (0.37/1.76). In the average system relative complexity metric, the Catalog microservice presented a distance of 50.25% (16.27/32.38). For the micro sales service, the calculated distance was 70.67% (30.58/43.27). By reducing complexity, ease of maintainability is achieved, so Mind Overflow is considered effective in terms of complexity.

Bugs. The average bug metric value per class has decreased for both microservices. For the Catalog and Sales microservice, the distance represented 50% (0.01/0.02 and 0.02/0.04). By reducing the metric, Mind Overflow is considered effective in terms of bugs.

6. CONCLUSIONS AND FUTURE WORKS

This paper presented Mind Overflow, which is an approach supported by a technology-agnostic process and framework to guide the decomposition of a monolithic application to a microservices architecture through a sequence of decomposition. Researchers and developers can benefit from the proposed approach to use it as a basis for new decomposition approaches as well as a technique for decomposing monolithic applications, respectively. The initial assessment of the approach through a case study showed promising results, in particular, it highlighted that the approach is feasible for decomposing monolithic to microservice-based architectures, including reducing cyclomatic complexity and producing highly cohesive microservices. As future work, we intend to conduct new case studies to improve the assessment and understand the strengths and improvements of the approach.

Finally, Mind Overflow is expected to serve as a basis for new work to emerge from it, serving as a starting point. Finally, we hope that the results discussed throughout this article can encourage practitioners to use the proposed approach. Besides, this work can be the first step for a more ambitious agenda on how to refactor monolithic applications into better-modularized ones.

7. ACKNOWLEDGMENTS

We also thank CNPq grant 313285/2018-7 for partially funding this research.

8. REFERENCES

[1] Keith Bennett. Legacy systems: Coping with success. *IEEE software*, 12(1):19–23, 1995.

[2] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87, 2000.

[3] Vinicius Bischoff, Kleinner Farias, Lucian José Gonçales, and Jorge Luis Victória Barbosa. Integration of feature models: A systematic mapping study. *Information and Software Technology*, 105:209–225, 2019.

[4] Alberto Brandolini. Introducing event storming. *Available at: goo.gl/GMzzDv [Last accessed: 8 July 2017]*, 2013.

[5] Leandro Ferreira D'Avila, Kleinner Farias, and Jorge Luis Victria Barbosa. Effects of contextual information on maintenance effort: A controlled experiment. *Journal of Systems and Software*, 159, 2020.

[6] Daniel Escobar, Diana Cárdenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra, and Rubby Casallas. Towards the understanding and evolution of monolithic applications as microservices. In *XLII Latin American Computing Conference (CLEI)*, pages 1–11. IEEE, 2016.

[7] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

Group	Metric	Catalog			Sales		
		Desired	**Generated**	**Dist.**	**Desired**	**Generated**	**Dist.**
LOC	Lines of code	760	1054	294	1305	1948	643
OOP	Classes	27	51	24	46	96	50
	Methods	80	109	29	130	195	65
Coupling	Afferent coupling average	0,48	1,06	0,58	0,67	1,43	0,76
	Efferent coupling average	1,85	2,25	0,4	2,13	2,66	0,53
Complexity	Average cyclomatic complexity per class	1,44	1,24	0,2	1,76	1,39	0,37
	Relative average system complexity	32,38	16,11	16,27	43,27	12,69	30,58
Bugs	Average bug per class	0,02	0,01	0,01	0,04	0,02	0,02

Table 3. : The obtained results.

[8] Kleinner Farias. Empirical evaluation of effort on composing design models. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 405–408. IEEE, 2010.

[9] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. Effects of stability on model composition effort: an exploratory study. *Software & Systems Modeling*, 13(4):1473–1494, 2014.

[10] Kleinner Farias, Alessandro Garcia, Jon Whittle, and Carlos Lucena. Analyzing the effort of composing design models of large-scale software in industrial case studies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 639–655. Springer, 2013.

[11] Martin Fowler and James Lewis. Microservices. *Viittattu*, 28:2015, 2014.

[12] Lucian José Gonçales, Kleinner Farias, Toacy Cavalcante De Oliveira, and Murilo Scholl. Comparison of software design models: An extended systematic mapping study. *ACM Computing Surveys (CSUR)*, 52(3):1–41, 2019.

[13] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

[14] Diane Kelly. A study of design characteristics in evolving software using stability as a criterion. *IEEE Transactions on Software Engineering*, 32(5):315–329, 2006.

[15] Holger Knoche and Wilhelm Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49, 2018.

[16] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016.

[17] Edson M Lucas, Toacy C Oliveira, Kleinner Farias, and Paulo SC Alencar. Collabrdl: A language to coordinate collaborative reuse. *Journal of Systems and Software*, 131:505–527, 2017.

[18] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[19] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[20] Anderson Oliveira, Vinicius Bischoff, Lucian José Gonçales, Kleinner Farias, and Matheus Segalotto. Brcode: An interpretive model-driven engineering approach for enterprise applications. *Computers in Industry*, 96:86–97, 2018.

[21] Diego Pereira da Rocha. Monólise: Uma técnica para decomposição de aplicações monolíticas em microsserviços. Master's thesis, 2018.

[22] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013.

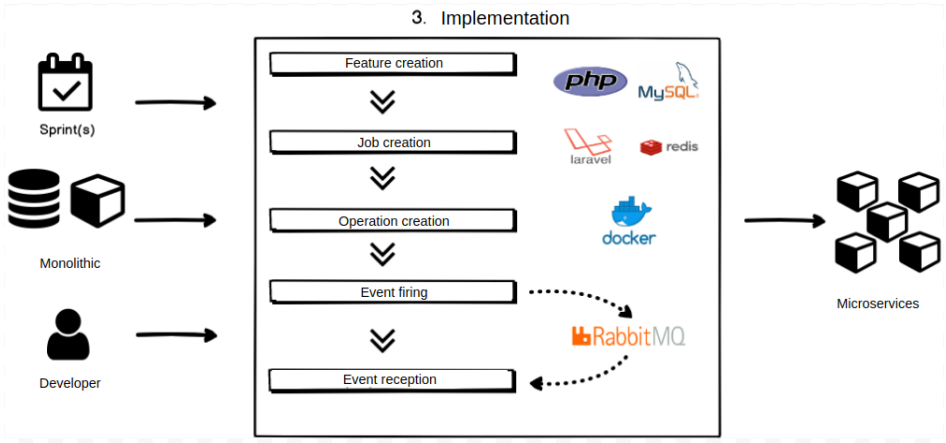

Fig. 5: Step 3: Implementation