Received: 26 October 2018 | Revised: 20 June 2019 | Accepted: 23 June 2019

DOI: 10.1002/spe.2733

RESEARCH ARTICLE

WILEY

Ranking open source application integration frameworks based on maintainability metrics: A review of five-year evolution

Rafael Z. Frantz | **Matheus H. Rehbein** | **Rodolfo Berlezi** | **Fabricia Roos-Frantz**

Department of Exact Sciences and Engineering, Unijuí University, RS, Brazil

Correspondence
Rafael Z. Frantz, Department of Exact Sciences and Engineering, Unijuí University, Ijuí, RS, Brazil.
Email: rzfrantz@unijui.edu.br

Funding information
Brazilian Co-ordination Board for the Improvement of University Personnel (CAPES), Grant/Award Number: 88881.119518/2016-01; Research Support Foundation of the State of Rio Grande do Sul (FAPERGS), Grant/Award Number: 17/2551-0001206-2

Summary

Integration frameworks are specialized software tools built and adapted to facilitate the design and implementation of integration solutions. An integration solution allows for the reuse of applications from the software ecosystem of companies to support their business processes. There are several open-source integration frameworks available on the market designed to operate in a business context to manipulate structured data; however, increasingly, they are required to deal with unstructured and large volumes of data, thus requiring effort to adapt these frameworks to work with unstructured and large volume of data. Choosing the framework, which is the easiest to be adapted, is not a trivial task. In this article, we review the newest stable versions of four open-source integration frameworks by analyzing how they have evolved regarding their adaptive maintainability over five years. We rank them according to their maintainability degree and compare past and current versions of each framework. To encourage and enable researchers and developers to replicate our experiments, with the aim of verifying our findings, and to experiment with new versions of the integration frameworks analyzed, we detail the experimental protocol used while also having made all the required software involved available on the Web.

KEYWORDS

adaptive maintenance, enterprise application integration, integration frameworks, integration patterns, integration platforms, software maintainability, software metrics

1 | INTRODUCTION

The software ecosystem[1] is gaining importance in the field of software engineering and has been investigated by the research community as an approach for software reuse.[2,3] It deals not only with technological aspects of how to connect applications but also with planning and keeping the development platform of companies under control; these platforms are now open to third-party software development companies that provide on-premise software and software as a service as well. The software ecosystem is composed of a diverse range of applications, which usually comprises on-premise applications, applications deployed to the cloud, software consumed as service from the cloud, and mobile applications that can be reused to support a business process. In this article, the term "software ecosystem" refers to a set of existing software applications running in an enterprise. These applications may be developed in-house by the enterprise's IT department, but it is common that the software ecosystem also includes off-the-shelf software packages purchased by the company. This set of applications is often heterogeneous because they can include applications developed in different

 © 2019 John Wiley & Sons, Ltd.

languages, running on different operating systems, etc. The diversity of programming languages, operating systems, development platforms, data models, etc, makes the communication between the applications within the software ecosystem difficult, mainly because most of them were not developed with integration in mind. This makes their integration for software reuse more complex.

Enterprise application integration (EAI) is a research field that deals with methodologies, techniques, and tools to support the development of integration solutions.[4] An integration solution orchestrates a set of applications aiming at the exchange of information and the reuse of functionality among the integrated applications.[5] Integration frameworks are specialized tools to develop integration solutions.[6,7] In recent years, several open-source message-based integration frameworks have emerged, which represent a new generation of application integration tools.[8,9] These frameworks follow the architectural style of *pipes-and-filters*[10] and are strongly influenced by the *integration patterns* documented by Hohpe and Woolf.[11] In the workflow of an integration solution, pipes represent message channels and filters represent atomic tasks that implement a concrete integration pattern to process encapsulated data in messages. The patterns document best practices to solve integration problems. In the EAI community, Apache Camel,[12] Spring Integration,[13] Mule ESB,[14] and Guaraná[15] are open-source message-based integration frameworks.

These integration frameworks were designed to operate in a business context to manipulate structured data, which flow inside an integration solution and are temporarily stored in channels that are used to desynchronize tasks in the workflow. Given that, in the pipes-and-filters architectural style, a message must be completely stored in a channel before being processed by the next task in the workflow, it is not adequate when the data are large or have to be processed in streaming. Increasingly integration frameworks are required to deal with unstructured and large volumes of data,[16] making the EAI research field interesting from a practical point of view,[5,8,9,17-19] whereas effort is required to adapt the integration frameworks for them to work with unstructured and large volumes of data.[17,20] A typical situation happens in contexts such as problems associated with natural language analysis, image analysis, video analysis, video-to-text, and extraction of text data in natural language, in which it is required to work with unstructured data that usually require a streaming pipeline.

Analyzing the maintainability of an integration framework is an important step toward its adaptation and is not an easy task. Adaptive maintenance is classified by Radatz et al[21] as a type of software maintenance. It focuses on adapting a software system, enabling the software to be used in contexts in which it was not developed for. Many researchers have proposed maintainability metrics that are related to the effort required to maintain and adapt a piece of software.[22-34] These metrics have been used to analyze and study software systems by avoiding practical experimentation with adaptation, which may result in additional costs. These metrics have been consolidated as powerful tools to provide data regarding software maintenance, which can then be used to find which framework requires less effort to be adapted to a specific context.

A methodology was proposed by Frantz et al[35] to analyze the maintainability of integration frameworks. The authors have organized it into the following steps: compute metrics, compute rank, check the rank, and rank pairs. The first step deals with the computation of 25 maintainability metrics from the literature to help software engineers to analyze the maintainability of EAI frameworks. These metrics were grouped into the following categories, based on the model proposed by Lanza and Marinescu[22]: size metrics, coupling metrics, complexity metrics, and inheritance metrics. In the second step, an empirical rank for each proposal regarding the analyzed metric is calculated. The third step statistically checks the rankings computed in the previous step. Iman-Davenport's test is used to check if the empirical rank can be statistically significant. The last step uses Bergmann-Hommel's test to compare each pair of proposals regarding their metrics and keeps the error rate of the comparisons under strict control.

In their proposal, the authors applied this methodology to check the maintainability of the following integration frameworks, in their respective versions: Apache Camel 2.7, Spring Integration 2.0, Mule ESB 3.1, and Guaraná 1.2. It is important to note that only the *core* code of those frameworks was considered because it is the only code to have the same functionality across all frameworks, since taking all the implemented code would be unfair as different frameworks have different adapters or components. In their proposal, it was possible to analyze and identify which of those versions of the integration frameworks required less effort in terms of its adaptation to a specific context.

In this article, we review the current stable versions of the same integration frameworks, which represent over five years of evolution of their source-code up to August 2017. The versions considered in this article are Apache Camel 2.17, Spring Integration 4.3, Mule ESB 3.8, and Guaraná 2.0. Please note that, in this article, our purpose is only to analyze the maintainability of these integration frameworks centered on the code of their *core* package, so that it can be used by software engineers as one more element in the process of making a decision regarding adaptation; however, it cannot be used as a single element to make the final decision, since this involves analyzing a variety of other factors, in addition to

maintainability, such as documentation, training courses, technical support, and the set of adapters, related to the integration framework. Thus, the contribution of this article is three-fold. First, we strictly apply the methodology proposed by Frantz et al[35] to these new versions of the integration frameworks to compute the new ranking based on maintainability and to check if their evolution over time has changed the position of the integration frameworks in the original raking computed by the authors. We also update information on the maintainability metrics and the effort required to adapt these integration frameworks to a specific context. We carefully discuss the new data and the new ranking. We have identified changes in the empirical rank and the ranking pairs of the integration frameworks. Second, we check the evolution of each maintainability metric in every integration framework by comparing the data we found in the current versions of these frameworks to those provided by Frantz et al.[35] We analyze by how much every metric has increased or decreased by comparing this data. We have observed that every integration framework has grown in size, which has made a direct impact on other metrics in coupling, complexity, and inheritance. Third, we introduce the experimentation protocol absent in the article by Frantz et al[35] while having made all the required software involved available on the Web, so as to encourage and enable researchers and developers to replicate our experiments with the aim of verifying our findings and to experiment with new versions of the integration frameworks analyzed.

The rest of this article is organized as follows. Section 2 provides background information on the integration frameworks and the maintainability metrics that were considered. Section 3 introduces in detail the experimentation protocol. Section 4 discusses the data collected for every metric of the current version of the integration frameworks. Section 5 presents the empirical rank and compares the different integration frameworks. Section 6 compares past and current versions of the same integration framework. Finally, Section 7 presents our conclusions.

2 | PRELIMINARIES

In this section, we provide a brief overview of the integration frameworks we have analyzed and introduce the maintainability metrics, which are the foundation of the applied methodology.

2.1 | Integration frameworks

Integration frameworks usually provide a common set of features to support the design, implementation, testing, execution, and monitoring of integration solutions. They often provide a domain-specific language, a software development kit, a testing environment, a monitoring tool, and a runtime system. The domain-specific language focuses on the elaboration of conceptual models for integration solutions, with an abstraction level close to the problem domain. The software development kit allows for transforming the conceptual models into an executable code. The environment for testing allows for running individual parts or the whole integration solution with the objective of identifying and eliminating possible bugs in the implementation. The monitoring tool is used to follow, at runtime, the behavior of the integration solution and to detect errors that could occur during the execution. The runtime system provides the full support necessary for the execution of those integration solutions.

Apache Camel is a Java-based integration framework, hosted by the Apache Software Foundation. This framework follows a code-centric development approach and provides a fluent API[36] in Java to implement the integration solution. Scala DSL or XML Spring-based configuration files can also be used to implement the solution. Apache Camel provides a tool that can be used to generate a graphical model that represents the integration solution only for visualizations. This model is represented using a domain-specific language that has its concrete syntax based on integration patterns. In Apache Camel, messages are processed by routes that have one or more inputs and one or more outputs. There is a library for exception handling to create policies for redelivering messages in case some exception occurs inside the routes. In addition, Apache Camel allows developers to analyze and test routes after their implementation by inspecting the routes' processing. During the execution, Apache Camel has no means to set a message priority or dynamically change the routes. During execution, it is possible to monitor the consumption of network, disk access, and memory usage by the integration framework; in addition, at the application level, Apache Camel provides special monitors that present information on the integration solution. There is also a commercial version of Apache Camel, which then provides a Web-based and a stand-alone Eclipse-based IDE, both with a visual editor to design integration solutions, called Fuse Source. This version is provided by the company Red Hat, based on the software-as-a-service model. At the time of writing this article, the last stable version launched was 2.17.

Spring Integration is another code-centric and Java-based integration framework, built on top of the Spring Framework container. As with any Spring-based application, integration solutions can be implemented using XML Spring-based

configuration files or a command-query API.[36] The textual domain-specific language provided by Spring Integration to implement integration solutions is based on integration patterns. This integration framework is provided as a stand-alone application and includes an Eclipse-based IDE with a graphical editor. In Spring Integration, messages can have different priorities, which allow those with a high priority to flow faster within the integration solution. Monitoring the integration solution running in Spring Integration requires third-party tools; only basic tools for monitoring memory and CPU consumption are offered by the integration framework. The new version of Spring Integration now includes connectors for many Web, Internet of Things, and cloud applications to build up solutions.

In common with previous frameworks, Mule ESB is another Java-based integration framework that implements enterprise service bus concepts. It is a project hosted by the company MuleSoft and follows a model-centric development approach, which means solutions can be designed using an intuitive visual editor based on an Eclipse IDE with drag-and-drop functionality. For those who prefer coding, it is also possible to use a command query API and XML Spring-based configuration files to implement integration solutions. The visual domain-specific language provided by Mule can raise the level of abstraction for designing integration solutions. It is simple to test integration solutions with Mule's own XML-based debug language.

Guaraná is a Java-based integration framework and follows a model-centric development approach. It provides an easy-to-learn and intuitive visual domain-specific language inspired by the integration patterns to design platform-independent models for integration solutions. It also provides a command-query API that can be used to implement integration solutions, although models designed using the visual language can be automatically transformed into executable code.

Messages that flow in an integration solution can have different priorities and their processing can be monitored by an external component, which can detect and analyze possible errors during message processing. The expected behavior of an integration solution can be expressed using a rule-based language, which allows the monitor to detect possible abnormal behavior. This monitoring also includes the possibility to observe the consumption of computational resources, such as memory and CPU. Guaraná is the result of a six-year joint effort between the academy and industry to provide new languages, methodologies, and tools to help integration engineers reduce the costs involved in the design and implementation of EAI solutions. At the time of writing this article, the last stable version of Guaraná was 2.0.

2.2 | Maintainability metrics

In this section, we introduce the 25 maintainability metrics, which comprise the methodology. Based on the work of Lanza et al,[22] these metrics were classified into four groups, namely, size, coupling, complexity, and inheritance. Size metrics can be used to indicate how big a software system is. Coupling metrics show the encapsulation degree of data and the collaboration of objects to perform system functionality. Complexity metrics show how complex it is to understand the source code. Inheritance metrics indicate how much and how well the concept of inheritance is used in a software system. All of these metrics can be automatically computed by using a specific kind of software, such as Metrics[37] and iPlasma.[22]

Size metrics

The metrics in this group represent how big the software is. Size metrics are represented by the number of packages, classes, interfaces, lines of code, attributes, methods, and parameters per method.

NOP: Number of packages that contain at least one class or interface. It is important to have a well-designed system, so this metric allows us to know how much effort is required to understand the organization of packages.[38] The greater this value, the more effort is required.

NOC: Number of classes. The source code of a software system implemented with an object-oriented language is composed of classes, so the more classes it has, the more difficult it becomes to understand its functionality.

NOI: Number of interfaces. The interfaces that comprise the software system are implemented by its classes. It is commonly agreed that the larger the number of interfaces, the easier it is to a adapt a software system.

LOC: Number of lines of code, excluding blank lines and comments. The more lines of code a software system contains, the more difficult it is to maintain that system.

NOM: Number of methods in classes and interfaces. This indicates the potential reuse of a class, as reported by Lorenz and Kidd[39] and by Chidamber and Kemerer,[31] where a large number of methods indicates that a class is likely to be application-specific, limiting the possibility of reuse.

NPM: Number of parameters per method. Methods that have a high number of parameters are harder to understand and frequently more complex. According to Henderson-Sellers,[32] the number of parameters should not exceed five. If it does, the author suggests that a new type must be designed to wrap the parameters into a unique object. The greater this value, the more difficult it is to understand a method.

MLC: Number of lines in methods, excluding blank lines and comments. For readability and maintainability reasons, Henderson-Sellers[32] recommends that the method should not exceed 50 lines; if it does, he suggests splitting the method into new smaller methods. The greater this value, the more difficult it is to understand and maintain a method.

NSM: Number of static methods. This metric indicates how well implemented a piece of code is. The greater this value, the more likely that the code tends to be based on the classical procedural paradigm and not on the object-oriented paradigm.

NSA: Number of static attributes. A large number of static attributes makes the process of reasoning about the state of a software system during tests difficult. The greater this value, the more difficult the testing.

NAT: Number of attributes. If a class has too many different attributes, understanding it becomes more complex. The greater this value, the more difficult it is to understand the state of a class.

Coupling metrics

These represent the main characteristic of the object-oriented paradigm, data encapsulation, and object collaboration necessary to perform system functionality. The metrics in this group give an indication of how the software system classes are coupled.

LCM: Lack of cohesion of methods. Cohesion refers to the number of methods that share common attributes in a single class, and the lack of cohesion is computed using the Henderson-Sellers LCOM* method.[32] A low value indicates a cohesive class, and a high value, close to 1, indicates a lack of cohesion, which suggests that the class might be split into two or more classes, because some methods might not belong to that class.

AFC: Afferent coupling. This is defined as the number of classes outside a package that depends on one or more classes inside that package.[33,40,41] The greater this value, the more complex maintenance becomes because there are more dependencies between classes, as well as indicating that the package is critical for the software system and that its maintenance must be done carefully, so as not to introduce problems for dependent classes.

EFC: Efferent coupling. The value of this metric is defined by the number of classes inside a package, which depends on a class outside the package.[33,40,41] The greater this value, the more likely that maintenance will have an impact on a package.

FAN: Number of called classes. According to Lorenz and Kidd,[39] this metric indicates how method calls are dispersed in a class. The greater this value, the more complex a method call is because every call is supposed to involve other classes to be completed.

LAA: Locality of attribute accesses. This metric represents how dependent a method of the attributes outside its class is. The greater this value, the more a method inside of a class uses external attributes.

CDP: Coupling dispersion. This metric represents how badly a method is written. The greater this value, the more likely that there is an improper distribution of functionality among the methods of a software system.

CIT: Coupling intensity. This metric can be used as an indicator of how dependent a method is, since it metrics the number of distinct methods that are called by the measured method. The greater this value, the more likely there is an excessive coupling among the methods of a software system.

Complexity metrics

The metrics inside this group show how complex a software system is and how complex and difficult it could be to understand its functionality and maintain it.

ABS: Degree of abstractness of a software system. This metric can be used as an indicator of how customizable a software system is.[33] The greater this value, the easier to customize the software system.

WMC: Weighted sum of McCabe cyclomatic complexity[34] for all methods in a class. This is an indication of how difficult it is to understand and modify the methods of a class.[31] The greater this value, the more effort is required to maintain a class.

MCC: McCabe cyclomatic complexity. This indicates how complex the algorithm in a method is, and this value should not exceed 10, according to McCabe.[34] The greater this value, the more difficult it is to maintain a piece of code.

WOC: Weight of class. This metric indicates the ratio of accessor methods regarding other methods that provide services.[42] The greater this value, the more class interfaces are used by accessor methods, indicating that classes are not too complex.

DBM: Depth of nested blocks in a method. This metric is used to indicate how expensive debugging a piece of code is. According to Henderson-Sellers,[32] this value should not exceed 5; if it does, he suggests that the method should be broken into other methods. The greater this value, the more complex an algorithm is.

Inheritance metrics

The main characteristic of an object oriented toward a paradigm is code reuse via the inheritance of functionality among classes. This allows us to know and understand how much and well applied the inheritance is in the software system looking at the following metrics.

DIT: Depth of inheritance tree. Inheritance is a mechanism to increase code reuse, and checking this metric enables us to know how complicated maintaining a class can be.[43] The greater this value, the more difficult it is to maintain a software system.

NOH: Number of immediate children classes of a class. A class can have an impact on a software system and, if it is modified, this metric indicates the potential impact.[31] The greater this value, the greater the chances that the abstraction defined by the parent class is poorly designed.

NRM: Number of overridden methods. Overridden methods adapt methods from their ancestors and this metric indicates how adaptable a class is concerning their ancestors.[39] The greater this value, the more likely that the inheritance mechanism is being used to adapt a class, instead of just providing additional services to the parent class.

3 | EXPERIMENTATION PROTOCOL

In this section, we present the research protocol we have used to review the current stable version of the integration frameworks analyzed by Frantz et al[35] and compute the new ranking. The protocol is composed of four main steps, namely, setting up the environment, collecting the core packages, computing metrics, and computing ranks. In each step, a set of detailed instructions is provided to realize it and make the experiment we carried out with the integration frameworks repeatable and extensible to other and newer versions of these frameworks. Figure 1 provides an overview of this protocol

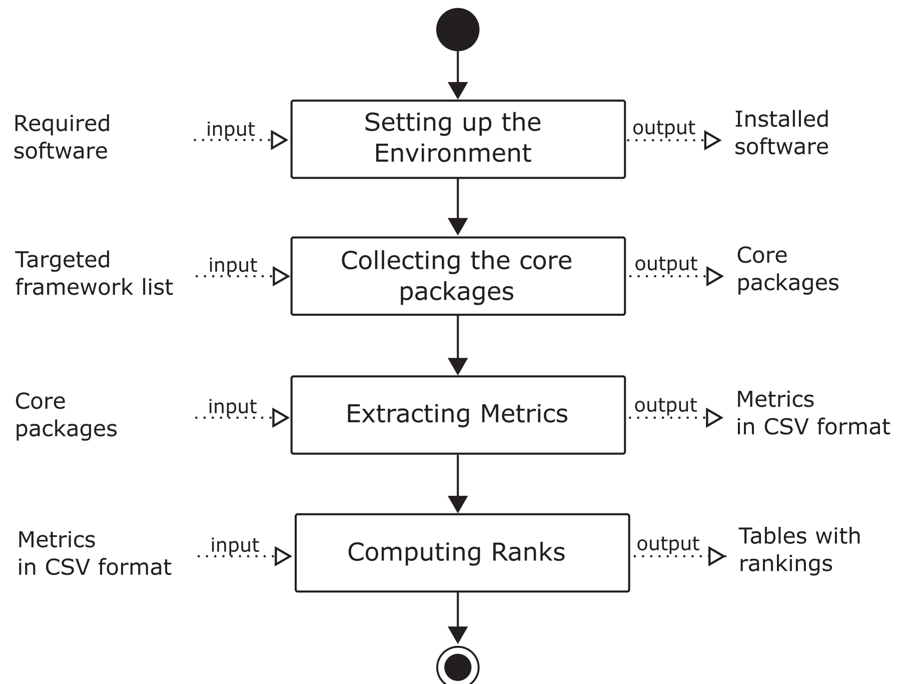

FIGURE 1 Overview of the protocol used to carry out the experimentation. CSV, comma-separated format

and highlights the input and output of each step, which are detailed in the following sections. Every piece of code and software tool required to carry out this experimentation protocol are made available on the Web to encourage and enable researchers and developers to replicate our experiments with the aim of verifying our findings and to experiment with new versions of the integration frameworks analyzed. The protocol we present here, together with the piece of code and software tools, can also be used by other researchers and developers to validate and improve the original methodology[35] on which we have based our study in this article.

3.1 | Setting up the environment

A set of five software tools must be downloaded and configured to carry out the experimentation. As Java is the technology under these tools, the experiment is platform-independent. This step takes, as input, the list of required software and, as outputs, these software packages installed. The instructions in the following guide the realization of this step.

1. Download and install Java SE version 7.0, which is required to run all the other software packages:
   https://www.oracle.com/technetwork/java/javase/
2. Download Eclipse Neon from the Eclipse Foundation website at:
   https://www.eclipse.org/downloads/
3. Follow the instructions at the Metrics 1.3.6 website to install and configure its plug-in into Eclipse:
   http://metrics.sourceforge.net
   Optionally, a preconfigured version of Eclipse Neon with a Metrics plug-in already installed can be downloaded and from the following link: http://www.gca.unijui.edu.br/publication/data/spe-a/eclipse-metrics.zip
4. Download and unzip iPlasma software in any local directory in the computer from the following link:
   http://www.gca.unijui.edu.br/publication/data/spe-a/iplasma.zip
5. Download and unzip MultipleTest software, which is required to run the statistical analysis in order to compute the rankings, in any local directory in the computer from the following link:
   http://www.gca.unijui.edu.br/publication/data/spe-a/multipletest-2.7.zip

3.2 | Collecting the core packages

The maintainability metrics are computed against the source code of the integration frameworks. This step takes, as input, a list with the names of the integration frameworks to be analyzed and, as outputs, individual and ready-to-be compiled Java Projects inside Eclipse, which contain only the core packages of each framework. This step can be realized by using the following instructions.

1. Inside Eclipse, create a separate and empty Java Project for each integration framework.
2. Download the corresponding complete source code for each integration framework from the following links:

   Apache Camel:
   https://github.com/apache/camel/tree/camel-2.17.0
   Spring Integration:
   https://github.com/spring-projects/spring-integration/tree/v4.3.0.RELEASE
   Mule ESB:
   https://github.com/mulesoft/mule/tree/mule-3.8.0
   Guaraná:
   http://www.gca.unijui.edu.br/publication/data/spe-a/guarana-2.0.zip

3. From the complete source code downloaded, copy only the Java classes of the "core" package of each integration framework by copying the directory "core" to the inside of the "src" directory of the corresponding Java Project.

   Apache Camel:
   Locate the directory "camel-core" and navigate to "src->main->java", and copy the directory "org" with all of its subdirectories to the inside of the "src" directory of the corresponding Java Project just created.
   Optionally, download the core at: http://www.gca.unijui.edu.br/publication/data/spe-a/camel-core.zip
   Spring Integration:
   Locate the directory "spring-integration-core" and navigate to "src->main->java", and copy the directory "org" with all of its subdirectories to the inside of the "src" directory of the corresponding Java Project just created.

Now, at Java Project, locate, from inside the "src", the package "org->springframework->integration->endpoint->management" and remove it. It contains a "package-info.java" file, which is just a placeholder file and thus an invalid Java class, which leads to an error when computing values for the metrics.
Optionally, download the core at: http://www.gca.unijui.edu.br/publication/data/spe-a/spring-integration-core.zip
Mule:
Locate the directory "core" and navigate to "src->main->java", and copy the directory "org" with all of its subdirectories to the inside of the "src" directory of the corresponding Java Project just created.
Optionally, download the core at: http://www.gca.unijui.edu.br/publication/data/spe-a/mule-core.zip
Guaraná:
Locate the directory "guarana-sources" and navigate to "guarana->guarana-framework->src" and copy the directory "guarana" with all of its subdirectories to the inside of the "src" directory of the corresponding Java Project just created. Now, locate the directory "guarana-sources" and navigate to "guarana->guarana-toolkit->src", and copy the directory "guarana" with all of its subdirectories to the inside of the same "src" directory of the corresponding Java Project just created.
Optionally, download the core at: http://www.gca.unijui.edu.br/publication/data/spe-a/guarana-core.zip

4. As Metrics does not work, if any class library is missing from the compilation of the core Java classes of the integration framework, it is required to import all the necessary libraries into each Java Project. Download a zip file containing all the necessary libraries to each framework from the following links:

   Apache Camel:
   http://www.gca.unijui.edu.br/publication/data/spe-a/camel-libs.zip
   Spring Integration:
   http://www.gca.unijui.edu.br/publication/data/spe-a/spring-integration-libs.zip
   Mule:
   http://www.gca.unijui.edu.br/publication/data/spe-a/mule-libs.zip
   Guaraná:
   http://www.gca.unijui.edu.br/publication/data/spe-a/guarana-libs.zip

Optionally, a preconfigured workspace of Eclipse, containing four Java Projects already with the only "core" package for each integration framework, as well as with the required libraries imported, can be downloaded from the following link: http://www.gca.unijui.edu.br/publication/data/spe-a/workspace.zip

3.3 | Computing Metrics

Metrics are computed individually for each integration framework by taking, as input, the core package of the framework. Both software packages have to be used because some metrics are computed with Metrics 1.3.6 and others are computed with iPlasma 6.1; however, there is no metric computed using both software packages. From the set of metrics, the following are computed by iPlasma: number of called classes (FAN), locality of attribute accesses (LAA), coupling dispersion (CDP), coupling intensity (CIT), and weight of class (WOC). Please, note that Table 1 is simply a collection of the metrics, whereas Table 4 only shows the increment/decrement percentage for each maintainability metric of the two integration frameworks' versions being compared. The following instructions have to be used for each framework at a time. Different to the majority of metrics that fall within the class of "the smaller the value of a metric, the better it is", the metrics NOI, ABS, WOC, and NRM fall within the class of "the smaller the value, the worse it is". To apply the statistical tests, it is necessary to keep every metric with the same goodness, ie, the smaller the value, the better it is. Thus, for Instructions 3 and 4 in the following, the values for NOI, ABS, WO, and NRM in the "Total" and "Mean" columns have to be normalized by subtracting them from their maxima, so that the comparison is homogeneous.

1. Metrics

   (a) The Metrics View has to be displayed to visualize the computed metrics. This can be performed in Eclipse by opening the menu "Windows->Show View->Other" and navigating to the "Metrics View".
   (b) The computing process starts by right-clicking on the Java Project, which contains the source code of the core package to be analyzed and via the pop-up menu by selecting "Metrics->Enable".

TABLE 1 Computed metrics for the last stable versions of Apache Camel, Spring Integration, Mule ES and Guaraná

	Metrics	Apache Camel 2.17				Spring Integration 4.3				Mule ESB 3.8				Guaraná 2.0			
		Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
Size	NOP	77	-	-	-	50	-	-	-	156	-	-	-	20	-	-	-
	NOC	1205	15.649	24.723	137	495	9.90	12.067	72	1184	7.590	11.034	86	96	4.80	2.891	11
	NOI	309	4.013	14.759	88	95	1.90	2.385	13	369	2.365	4.385	27	13	0.650	1.236	4
	LOC	119 418	-	-	-	33 139	-	-	-	105 077	-	-	-	3504	-	-	-
	NOM	12 905	10.71	17.438	274	3086	6.234	6.780	53	8309	7.018	10.456	140	435	4.531	4.684	33
	NPM	-	0.932	1.147	14	-	1.018	1.063	9	-	0.945	1.104	20	-	1.193	1.038	4
	MLC	67 024	4.761	9.554	150	18 360	5.795	10.974	145	54 584	5.939	10.177	209	2107	4.833	6.499	54
	NSM	1185	0.983	5.412	102	82	0.166	1.430	29	883	0.746	8.155	266	1	0.010	0.102	1
	NSA	553	0.459	1.273	18	224	0.453	1.517	21	964	0.814	3.990	119	35	0.365	1.217	10
	NAT	3657	3.035	5.179	85	1056	2.133	3.175	20	2286	1.931	3.144	35	106	1.104	2.119	13
Coupling	LCM	-	0.314	0.363	1	-	0.252	0.342	1	-	0.233	0.327	1.333	-	0.136	0.264	0.918
	AFC	-	38.974	126.029	895	-	12.40	13.393	60	-	27.590	75.853	766	-	8.40	17.582	61
	EFC	-	15.429	23.628	119	-	9.90	11.942	71	-	7.628	9.397	54	-	4.80	2.60	11
	FAN	7980	4.720	-	113	1380	1.950	-	54	6193	3.670	-	145	183	1.560	-	11
	LAA	14 474.04	0.960	-	1	3290.44	0.980	-	1	9874.83	0.980	-	1	447.25	0.960	-	1
	CDP	2045.17	0.140	-	1	345.66	0.10	-	1	1580.60	0.160	-	1	32.60	0.070	-	1
	CIT	5396	0.360	-	42	655	0.20	-	22	3748	0.370	-	30	75	0.160	-	7
Complexity	ABS	-	0.163	0.258	1	-	0.318	0.261	1	-	0.357	0.333	1	-	0.531	0.332	1
	WMC	23 915	19.846	32.789	556	5747	11.610	14.392	107	16 036	13.544	21.481	282	594	6.188	6.379	47
	MCC	-	1.699	2.353	64	-	1.814	2.295	33	-	1.745	1.859	39	-	1.362	0.947	8
	WOC	1048.28	0.620	-	1	378.01	0.530	-	1	1106.03	0.660	-	1	77.21	0.660	-	1
	DBM	-	1.344	0.759	8	-	1.399	0.898	7	-	1.419	0.847	8	-	1.245	0.734	4
Inheritance	DIT	-	2.271	1.325	7	-	2.317	1.504	7	-	2.225	1.476	7	-	3.083	1.304	5
	NOH	812	0.674	4.685	97	229	0.463	1.457	12	588	0.497	1.881	26	63	0.656	1.875	10
	NRM	905	0.751	1.165	8	207	0.418	1.021	13	500	0.422	0.982	10	85	0.885	0.999	3

2. iPlasma
    (a) As this software does not require installation, localize the directory in which iPlasma was unzipped and navigate to the directory "tools->iPlasma" and run "insider.bat" (Windows) or "insider.sh" (Linux), according to the running operating system. This will open the main window of iPlasma.
    (b) The directory containing the core package of every integration framework has to be loaded individually to compute the metrics. This can be carried out by clicking on "load->java sources->source path" and selecting the directory.
    (c) The source code is loaded under "~root", which must now be right-clicked to select "select columns". This action opens a new window where metrics can be aggregated by localizing them via the "search" field.
    (d) The search field allows one metric (called property) per time to be aggregated. Once the metric is located, select it and, in the next box, aggregate the columns "avg", "sum", and "max", so that the corresponding values are shown in the main window of iPlasma. The following properties in this software package correspond to the metrics in our article class_FANOUT (FAN), method_LAA (LAA), method_CDISP (CDP), method_CINT (CIT), and class_WOC (WOC).

3. Using a plain text editor such as Notepad, create an empty text file with the name "total.csv". Input the total values for every metric of all integration frameworks to this file. The first column must be named "TECH", and the other column "Total". Each row provides the value for one metric in a comma-separated format (CSV), such as in the following excerpt:

```
TECH,      Total
Camel,     77
Camel,     1205
Camel,     60
...
Mule,      156
Mule,      1184
Mule,      0
...
Spring,    50
Spring,    495
Spring,    274
...
Guarana,   20
Guarana,   96
Guarana,   356
...
```

   Optionally, a complete and ready-for-use "total.csv" file can be downloaded from the following link:
   http://www.gca.unijui.edu.br/publication/data/spe-a/total.csv.zip

4. Create another empty text file with the name "mean.csv". Input the mean values for every metric of all integration frameworks to this file. The first column must be named "TECH", and the other column "Mean". Each row provides the value for one maintainability metric of each framework in a comma-separated format (CSV), such as in the following excerpt:

```
TECH,      Mean
Camel,     15.649
Camel,     0
Camel,     10.71
...
Mule,      7.59
Mule,      1.648
Mule,      7.018
...
Spring,    9.9
Spring,    2.133
```

```
Spring,   6.234
...
Guarana,  4.8
Guarana,  3.363
Guarana,  4.531
...
```

Optionally, a complete and ready-for-use “mean.csv” file can be downloaded from the following link: http://www.gca.unijui.edu.br/publication/data/spe-a/mean.csv.zip

3.4 | Computing ranks

The last step is the computation of the raking, which takes, as input, the metrics in the CSV file format and, as outputs, the tables of the ranking. This step is supported by MultipleTest software, which must be executed against the mean and the total values. The following instructions guide the realization of this step and provide data for Figure 2 and Tables 2 and 3.

1. Move the CSV files containing the computed values for the metrics (total.csv and mean.csv) to the inside of the local directory where the MultipleTest software was unzipped.

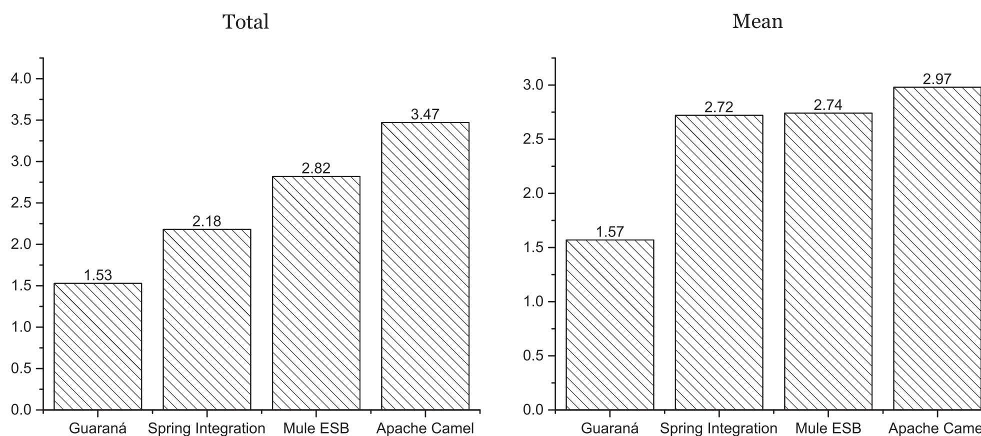

FIGURE 2 Empirical ranking of the integration frameworks

TABLE 2 Iman-Davenport's test

Test	Total	Mean
Statistic	11.52	7.00
P-Value	8.27×10^{-6}	3.69×10^{-4}

TABLE 3 Results of Bergmann-Hommel's test

(a) Total values				
Comparison	**Statistic**	**AP-Value**	**Integration Framework**	**Rank**
Apache Camel vs. Guaraná	4.384	0.287	Guaraná, Spring Integration, Mule ESB, Apache Camel	1
Mule ESB vs. Guaraná	2.923	0.287		
Apache Camel vs. Spring Integration	2.923	0.010		
Spring Integration vs. Mule ESB	1.461	0.287		
Spring Integration vs. Guaraná	1.461	0.287		
Apache Camel vs. Mule ESB	1.461	0.287		
(b) Mean values				
Comparison	**Statistic**	**AP-Value**	**Integration Framework**	**Rank**
Apache Camel vs. Guaraná	3.712	1.000	Guaraná	1
Mule ESB vs. Guaraná	3.084	1.000	Spring Integration, Mule ESB, Apache Camel	2
Apache Camel vs. Spring Integration	0.685	1.000		
Spring Integration vs. Mule ESB	0.057	1.000		
Spring Integration vs. Guaraná	3.027	0.006		
Apache Camel vs. Mule ESB	0.628	1.000		

2. Open a command prompt and navigate to the local directory where the MultipleTest software was unzipped. Inside this directory, run the following commands to compute ranks:

Compute the Total values to the ranking:
```
java -jar multiple-test-2.7.jar total.csv Total -
```
Compute the Mean values to the ranking:
```
java -jar multiple-test-2.7.jar mean.csv Mean -
```

4 | COMPUTED METRICS

The set of 25 maintainability metrics was computed for Apache Camel, Spring Integration, Mule ESB, and Guaraná, as suggested in the methodology and by following the protocol previously introduced. Table 1 summarizes the results that we collected.

The structural size of the integration frameworks was organized into packages: 77 in Apache Camel, 50 in Spring Integration, 156 in Mule ESB, and 20 in Guaraná. Even though Mule ESB has so many packages, Apache Camel has even more classes than Mule ESB, with 1205 and 1184 classes, respectively; in contrast, in Spring Integration, there are 495 classes, whereas Guaraná has only 96 classes. The maximum number of classes in a package is 137 for Apache Camel, 86 for Mule ESB, and 72 for Spring Integration, that is, almost half that of Apache Camel; meanwhile, Guaraná has just 11. A different situation occurs for the number of interfaces. Although Apache Camel has 309 interfaces in its packages, Mule ESB has 369, which indicates that Mule ESB is more adaptable. In Spring Integration, there are 95 interfaces, and in Guaraná, there are 13.

The number of classes also implicates the number of lines of code; and, as we can see, in Table 1, Apache Camel has 119 418 lines of code, Mule ESB has 105 077, Spring Integration has 33 139, and Guaraná has 3504. Thus, Apache Camel has almost four times more lines of code than Spring Integration, which has nine times more than Guaraná. These differences can also be seen in the number of methods. Apache Camel has a total of 12 905, with an average of 10.710, methods per class, Guaraná has only 435, Spring Integration has 3086, and Mule ESB has 8309, with mean values of 4.531, 6.234, and 7.018, respectively. Now, if we compare the maximum number of parameters per method, another important difference can be noticed: 14 in Apache Camel, 20 in Mule ESB, 9 in Spring Integration, and 4 in Guaraná. This indicates, especially in Mule ESB and Apache Camel, that there is much more complexity in some methods, which are also more limited, less reusable, and much harder to understand. So far, these methods are difficult to maintain. These differences can be observed again in the number of lines in methods: 150 lines in Apache Camel, 145 lines in Spring Integration, 209 lines in Mule ESB, and 54 lines in Guaraná, with a total number of lines in the methods equal to 67 024, 18 360, 54 584, and 2107, respectively, ensuring that methods in Guaraná are smaller and probably less complex to understand and maintain.

Regarding the number of static methods, we can see 1185 in Apache Camel, 883 in Mule ESB, 82 in Spring Integration, and only 1 in Guaraná. This is a significant difference, as a static system is more complex to adapt. In addition to the number of static attributes, there are 553 in Apache Camel, 224 in Spring Integration, 964 in Mule ESB, and 35 in Guaraná. Even regarding the total number of attributes, we can see significant asymmetry, where Apache Camel has about 3657 attributes and a maximum of 85 in a single class, whereas Spring Integration has 1056 attributes and a maximum of 20 in a single class, Mule ESB has 2286 attributes and a maximum of 35 in a single class, and Guaraná with 106 attributes and a maximum of 13 in a single class.

For coupling values, it is noticeable that the mean and the maximum values for the lack of cohesion among methods are very similar in every framework. These means are 0.314, 0.252, 0.233, and 0.136, with a maximum of 1, 1, 1.333, and 0.918, for Apache Camel, Spring Integration, Mule ESB, and Guaraná, respectively. Regarding the lack of cohesion, the closer the value is to 1, the more likely it is to split the class, so Mule ESB exceeds the limit. In addition, we have the values for afferent and efferent coupling classes in the tools. First, the afferent values of Apache Camel are very high, with a mean of 38.974 and a maximum of 895, whereas Spring Integration has 12.400 and 60, Mule ESB has 27.590 and 766, and Guaraná has 8.400 and 61 for the mean and the maximum, respectively. Observe how proximate Spring Integration and Guaraná are. Although the efferent value for Mule ESB is better than that of Apache Camel and Spring Integration, with a mean of 7.628 and a maximum of 54, it is still not as good as it could be. The efferent values of the mean and the maximum for Apache Camel are 15.429 and 119, whereas, for Spring Integration, they are 9.900 and 71. Guaraná stands out with values of 4.800 and 11. These data suggest that much more attention must be paid when performing maintenance on any classes of a package because the classes might have a high number of dependencies.

For the number of called classes, we have 7980 for Apache Camel, 1380 for Spring Integration, 6193 for Mule ESB, and only 183 for Guaraná. The locality of attribute accesses reveals the dependence of a method on attributes outside its class: this is 0.960 for Apache Camel, 0.980 for Spring Integration, 0.980 for Mule ESB, and 0.960 for Guaraná, thereby keeping every framework at the same level. Coupling dispersion indicates how properly distributed the methods are, with the mean values being 0.140 for Apache Camel, 0.100 for Spring Integration, 0.160 for Mule ESB, and 0.070 for Guaraná, which indicates that Mule ESB is the most dispersed. Regarding coupling intensity, the maximum dependency values are 42, 22, 30, and 7 in a method for Apache Camel, Spring Integration, Mule ESB, and Guaraná, respectively, demonstrating an excessive coupling, especially in the case of Apache Camel.

The degree of abstractness highlights how abstract an integration framework is, which reflects how easily it can be customized. Apache Camel has the lowest degree of abstractness, with a mean of 0.163, whereas it is 0.318 for Spring Integration, 0.357 for Mule ESB, and 0.531 for Guaraná. As it gets close to the maximum value of 1, the better it is. The weighted sum method of McCabe cyclomatic complexity for all methods in a class demonstrates a high complexity within that class. As we can see in Apache Camel, the sum is 23 915, whereas it is 5747 in Spring Integration, 16 036 in Mule ESB, and only 594 in Guaraná. The mean and the maximum values are 19.846 and 556 for Apache Camel, whereas they are 11.610 and 107 for Spring Integration, 13.544 and 282 for Mule ESB, and 6.188 and 47 for Guaraná. Thus, the weighted sum also implicates McCabe cyclomatic complexity, which is high for these frameworks. They reached values of 64, 33, and 39 for Apache Camel, Spring Integration, and Mule ESB, respectively. Thus, all of them have at least three times more than the recommended value, which is 10. Those values indicate that the frameworks are complex and that their maintenance could be difficult. Guaraná reached a maximum complexity of 8.

Regarding the weight of class, we obtained mean values of 0.620, 0.530, 0.660, and 0.660 for Apache Camel, Spring Integration, Mule ESB, and Guaraná, respectively. This maintainability metric indicates that the greater this value is, the more complex the classes are, in terms of the ratio of accessing methods. Considering the maximum value for the depth of nested blocks in a method, Apache Camel and Mule ESB have 8, whereas Spring Integration has 7 and Guaraná has 4. The cost of debugging a piece of code is more expensive in the cases of Apache Camel and Mule ESB.

Finally, for inheritance parameters, the depth of the inheritance tree is 7 for every framework, except for Guaraná, whose value is 5. Therefore, the maximum number of immediate children classes has significant differences: 97, 12, 26, and 10 for Apache Camel, Spring Integration, Mule ESB, and Guaraná, respectively. These values suggest that the parent classes of Apache Camel were poorly designed, making it harder to carry out proper maintenance without having compatibility problems. Still, the four frameworks have a maximum number of overridden methods of 8 for Apache Camel, 13 for Spring Integration, 10 for Mule ESB, and 3 for Guaraná concerning a single method.

5 | STATISTICAL ANALYSIS

When analyzing a set of data values by using the mean measure, it may lead to wrong conclusions, since, for skewed distributed values, the mean is not necessarily the best choice. For example, a uniformly distributed set of numbers may have the same mean as a highly skewed set, but they behave quite differently in practice. Analyzing the standard deviation in addition to the mean could help; but then, the problem becomes how to compare two different indicators at the same time. This comparison is questionable because it is difficult to guarantee that the empirical data support the hypothesis that the differences are statistically significant. Many authors from the statistics field have been motivated to approach nonparametric tests[44] for computing and comparing the empirical ranks of the metrics instead of their values. What happens is that these tests are completely independent of the distribution of the values, making these kinds of tests more resistant to outliers, such that there is a better chance that there are no wrong conclusions about the collected values. In the following sections, we compute and check the rank, as well as carry out a pairwise comparison of the analyzed integration frameworks following the methodology proposed by Frantz et al.[35]

5.1 | Computing rank

This rank allows us to see which integration framework has the best overall maintainability. As can be observed in Figure 2, the values of the empirical ranking of the integration frameworks indicate that both the total values and the mean values from Guaraná are better, with 1.53 and 1.57, respectively, followed by Spring Integration with 2.18 and 2.72, Mule ESB with 2.82 and 2.74, and Apache Camel with 3.47 and 2.98.

5.2 | Checking rank

The ranking is checked using Iman-Davenport's test, as shown in Table 2. It is necessary to determine if the differences in the empirical ranking from the last section have a significant value. In order to verify the existence of this meaningful statistic difference between the empirical values, it is important that a confidence interval of $\alpha = 0.05$ is used. A P-value is used to determine the significance of the results in a statistical test, whereas a small value, such as the interval of confidence which we use, could indicate strong evidence against the null hypothesis.

It is possible to see, in Table 2, that the P-value of the total values and the mean values are smaller than the statistic values, meaning that these values become irrelevant from a statistical point of view. Noting that the total P-value is smaller than the standard level, we are confident that the empirical ranks are different from a statistical point of view; thus, it becomes necessary to apply Bergmann-Hommel's test to rank every pair of frameworks.

5.3 | Ranking pairs of proposals

To rank the pairs of proposals, it is necessary to compare every pair regarding a metric. In order to obtain a static ranking, we have used Bergmann-Hommel's test.

As presented in Table 3, the adjusted P-values (AP-values) result from the comparison of Guaraná with the other three frameworks: Apache Camel, Mule ESB, and Spring Integration. When observing the total values for statistics, it is possible to see that the difference in maintainability between Guaraná and Apache Camel is the highest (4.384). The difference in maintainability between Guaraná and Mule ESB is also high (2.923). The maintainability of Guaraná is closer to the maintainability of Spring Integration (1.461). The mean values for statistics between Spring Integration and Mule ESB are very small (0.057), which suggests that the maintainability of these integration frameworks is almost the same. The difference in the mean values of Guaraná and Apache Camel is the highest (3.712), which reinforces the claim that they are the most different in terms of maintainability. The Bergmann-Hommel's test for the total values regarding each pair of proposals ranks the four integration frameworks at the same level. However, when analyzing the mean values in this test, Guaraná is ranked in the first place, followed by Spring Integration, Mule ESB, and Apache Camel in equal second place.

6 | COMPARING VERSIONS

In this section, we analyze how the source code of each integration framework evolved over the course of five years regarding its maintainability. To do so, we compare the maintainability data, which we computed for each metric of Apache Camel 2.17, Spring Integration 4.3, Mule ESB 3.8, and Guaraná 2.0, with the maintainability data originally computed by Frantz et al[35] for Apache Camel 2.7, Spring Integration 2.0, Mule ESB 3.1, and Guaraná 1.2. Table 4 summarizes the differences between values in each metric of the two compared versions. The values shown in this table indicate the increment (+) or decrement (−) percentages in each metric. Cells highlighted in dark gray indicate that there was an improvement in the last version of the integration framework with respect to this metric, whereas cells highlighted in light gray indicate that the last version of the integration framework worsened.

Starting with the size of metrics, the number of packages has increased over time in every framework. Spring Integration has the highest rate of increase at 56.25%, followed by Apache Camel at 42.59%, Mule ESB at 25.81%, and Guaraná at 11.11%. The number of classes was also affected: Spring Integration increased its number of classes by 84.01%, Apache Camel by 65.07%, Mule ESB by 61.53%, and Guaraná by 21.52%. This is also expected for interfaces, but in a positive way. Guaraná increased the number of interfaces by 44.44%, Mule ESB by 76.56%, Apache Camel by 120.71%, and Spring Integration by 137.50%. It is important to note that the more interfaces a system has, the more adaptable it is.

The lines of code also increased in every framework as expected, with Apache Camel and Spring Integration basically doubling their size by 91.26% and 121.98%, respectively. Looking at the number of methods, Spring Integration again is the one that increased the most by 115.65%, followed by Apache Camel by 83.96%, Mule ESB by 61.09%, and Guaraná by 17.89%. It is noteworthy that the mean value for Mule ESB decreased by 0.31% and, for Guaraná, it decreased by 2.98%, which is good. As for the number of parameters per methods, Guaraná and Spring Integration decreased their means by 0.58% and 9.91%, whereas Mule ESB and Apache Camel increased by 2.72% and 0.22%, respectively. The number of lines in a method increased, as a total and as a mean, for Apache Camel by 92.38% and 5.33%, for Spring Integration by 122.17% and 2.57%, and for Guaraná by 20.54% and 2.39%. It is noticeable how Mule ESB increased by 51.67% in total, whereas its mean decreased by 3.59%.

TABLE 4 Increment/decrement percentages of each maintainability metric for the two integration frameworks versions compared. Dark gray represents improvement; light gray represents degradation

	Metrics	Apache Camel (version 2.7 against 2.17)				Spring Integration (version 2.0 against 4.3)				Mule ESB (version 3.1 against 3.8)				Guaraná (version 1.2 against 2.0)			
		Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
Size	NOP	+42.59	-	-	-	+56.25	-	-	-	+25.81	-	-	-	+11.11	-	-	-
	NOC	+65.07	+15.75	+26.46	+42.71	+84.01	+17.72	+14.71	+24.14	+61.53	+28.43	+49.11	+68.63	+21.52	+9.34	-6.44	0
	NOI	+120.71	+54.94	+62.72	+51.72	+137.50	+52.00	+29.62	+44.44	+76.56	+39.94	+33.69	+50.00	+44.44	+30.00	+62.63	+100.00
	LOC	+91.26	-	-	-	+121.98	-	-	-	+56.62	-	-	-	+21.75	-	-	-
	NOM	+83.96	+11.45	+13.53	+42.71	+115.65	+17.18	+21.07	+35.90	+61.09	-0.31	+2.21	+10.24	+17.89	-2.98	+1.61	+37.50
	NPM	-	+0.22	+9.24	+27.27	-	-9.91	+13.09	0	-	+2.72	+3.18	+5.26	-	-0.58	-0.19	0
	MLC	+92.38	+5.33	+17.23	+6.38	+122.17	+2.57	+14.43	+31.82	+51.67	-3.59	-7.40	+16.11	+20.54	+2.39	+1.07	0
	NSM	+67.14	+1.34	+9.33	+37.84	+168.52	+38.33	+95.89	+262.50	+28.72	-20.64	-13.34	+9.02	0	0	-7.27	0
	NSA	+90.03	+14.75	+18.97	+12.50	+105.50	+10.49	-0.20	+61.54	+44.10	-10.55	+9.02	+46.91	+16.67	-3.95	-8.50	0
	NAT	+102.83	+22.87	+24.20	+37.10	+122.78	+21.19	+26.49	+25.00	+61.33	+0.05	-2.06	+12.90	+21.84	+0.36	-0.98	+8.33
Coupling	LCM	-	+8.28	+3.71	0	-	+14.55	+3.64	+6.38	-	+1.30	-3.82	+0.23	-	-2.86	-2.22	+0.88
	AFC	-	+27.24	+41.07	+65.13	-	-2.29	-49.74	-58.90	-	+20.48	+34.85	+55.38	-	+21.40	+22.69	+29.79
	EFC	-	+23.04	+32.52	+36.78	-	+17.30	+21.36	+29.09	-	+22.64	+39.01	+42.11	-	+15.11	-7.47	0
	FAN	+119.41	+26.20	-	+52.70	+114.95	+12.72	-	+35.00	+64.49	+1.94	-	+123.08	+4.57	+1.30	-	0
	LAA	+98.72	-1.03	-	0	+131.54	0	-	0	+57.87	0	-	0	+3.91	+1.05	-	0
	CDP	+133.80	+16.66	-	0	+177.42	+11.11	-	0	+68.15	+6.67	-	0	-7.91	-12.5	-	0
	CIT	+132.59	+16.13	-	+20.00	+156.86	+11.11	-	+15.79	+64.89	+5.71	-	0	+1.35	0	-	+16.67
Complexity	ABS	-	+8.66	+22.86	0	-	+17.78	+4.40	0	-	+8.18	+0.91	0	-	-1.67	-5.14	0
	WMC	+85.34	+12.25	+19.80	+ 60.69	+118.68	+18.83	+27.70	+57.35	+52.19	-5.81	-2.00	+7.63	+19.28	-1.78	+1.25	+27.03
	MCC	-	+1.74	+14.22	+39.13	-	+0.78	+12.50	+10.00	-	-3.06	-7.51	+18.18	-	+0.89	+4.07	0
	WOC	+80.36	+3.33	-	0	+111.85	+10.42	-	0	+67.88	+4.76	-	0	+4.20	+1.54	-	0
Inheritance	DBM	-	-1.90	-3.92	0	-	-2.85	+4.42	+16.67	-	-0.77	-2.64	0	-	+0.40	-0.81	0
	DIT	-	+2.30	-0.38	+16.67	-	-5.43	+7.43	+16.67	-	+10.15	+13.54	0	-	+1.75	-2.69	0
	NOH	+64.71	-0.88	+24.27	+40.58	+55.78	-15.82	-5.39	+9.09	+74.48	+8.04	+3.35	-7.14	+6.78	-12.53	-8.54	0
	NRM	+153.50	+53.27	+9.91	0	+200.00	+60.77	+54.70	+160.00	+42.45	-13.88	-3.73	+11.11	+21.43	-0.56	-1.09	0

The number of static methods did not change for Guaraná. For Spring Integration, it increased by 168.52%, whereas, for Apache Camel, it increased by 67.14% and, for Mule ESB, it increased by 28.72%. Mule ESB also decreased the mean value by 20.64%. For the number of static attributes, Mule ESB decreased its mean value by 10.55%, whereas Guaraná decreased the mean value by 3.95%, but increased its total value by 16.67%. Spring Integration increased its maximum value by 61.54%. Apache Camel increased all its values for this metric. At last, regarding the number of attributes, Spring Integration and Apache Camel showed the greatest total increase, with 122.78% and 102.83%, respectively. They were followed by Mule ESB with 61.33% and Guaraná with 21.84%.

Coupling metrics offer us another perspective on how elaborated the system is. For the lack of cohesion among methods, the mean values portray an increase of 14.55% for Spring Integration, 8.28% for Apache Camel, and 1.30% for Mule ESB, and a decrease of 2.86% for Guaraná. Guaraná also decreased by 2.86% and 2.22% for mean and deviation values. Afferent coupling decreased significantly only in the case of Spring Integration, with 58.90% in the maximum value and 49.74% in its deviation; all the other frameworks have increasing values. Concerning efferent coupling, Guaraná is the only framework that decreases a value by 7.47% in deviation, whereas there are no changes in the maximum value. The other three frameworks increased their values.

The total value for the number of called classes increased for Apache Camel by 119.41%, Spring Integration by 114.95%, Mule ESB by 64.49%, and Guaraná by 4.57%, which was the only framework that did not change in the maximum value. In contrast, the mean value for the locality of attribute accesses decreased in Apache Camel by 1.03% and increased in Guaraná by 1.05%, whereas, for Spring Integration and Mule ESB, there were no changes. For coupling dispersion, we found the highest increase in the total value in Spring Integration at 177.42%, followed by Apache Camel at 133.80% and Mule ESB at 68.15%; meanwhile, Guaraná decreased by 7.91%. The same applies to coupling intensity in which Spring Integration had an increase of 156.86% in its total value, Apache Camel had an increase of 132.59%, Mule ESB had an increase of 64.89%, and Guaraná had an increase of 1.35%. Note that Mule ESB had no change in the maximum value for coupling intensity.

Complexity metrics are essential to evaluate the maintainability of a system: the less abstract and complex the system is, the easier it is to understand and thus maintain it. The abstractness of Guaraná decreased by a mean of 1.67%, compared to an increase for Apache Camel and Mule ESB by 8.66% and 8.18%, respectively. The highest increase in abstractness was found in the case of Spring Integration, with a value of 17.78%.

The weighted sum of the McCabe cyclomatic complexity total value increased for Apache Camel, Sprint Integration, Mule ESB, and Guaraná by 85.34%, 118.68%, 52.19%, and 19.28%, respectively. It should also be acknowledged that Mule ESB and Guaraná are the only frameworks that decreased their mean value, by 5.81% and 1.78%, respectively. Guaraná also decreased its maximum value by 27.03%, which is important for maintainability. McCabe cyclomatic complexity increased the maximum value in all frameworks, with Apache Camel leading with 39.13%, followed by Mule ESB with 18.18%, Spring Integration with 10.00%, and Guaraná without any change.

The weight of a class not only significantly increased in Spring Integration, by a total of 111.85% and by a mean of 10.42%, but also in the others frameworks. The last metric of complexity is the depth of nested blocks in a method, where Apache Camel, Mule ESB, and Guaraná experienced no change in the maximum value, with only Spring Integration increasing its maximum value by 16.67%. Inheritance metrics can reveal how much code has been reused during the evolution of the versions. Both Apache Camel and Spring Integration increased the maximum value in the depth of the inheritance tree by 16.67%, whereas Mule ESB and Guaraná saw no changes. The total value of the number of immediate children classes of a class increased for every framework. For Apache Camel, Spring Integration, and Mule ESB, it increased by 64.71%, 55.78%, and 74.48%, respectively, whereas, for Guaraná, it only increased by 6.78%. Nevertheless, the mean value decreased for all frameworks, except for Mule ESB, which increased by 8.04%, but the maximum value decreased by 7.14%. Spring Integration and Apache Camel increased this metric's maximum value by 9.09% and 40.58%, respectively, whereas Guaraná had no change.

For the number of overridden methods, Spring Integration is the single framework, which increased all its values by 200.00% in total, and 60.77% and 160% in mean and maximum values. Apache Camel increased its total and mean values by 153.50% and 53.27%, whereas there were no changes in the maximum value. Mule ESB decreased the mean value by 13.88%, whereas the total and maximum values increased by 42.45% and 11.11%. Guaraná increased its total value by 21.43%, whereas the mean value decreased by 0.56%; the maximum value did not change.

A comparison of the empirical ranking computed for past and current versions of the integration frameworks analyzed is shown in Figure 3. In the new empirical ranking, Mule ESB moved up one position to occupy the third place, whereas Apache Camel dropped one position. Guaraná and Spring Integration retained the same position in the empirical ranking.

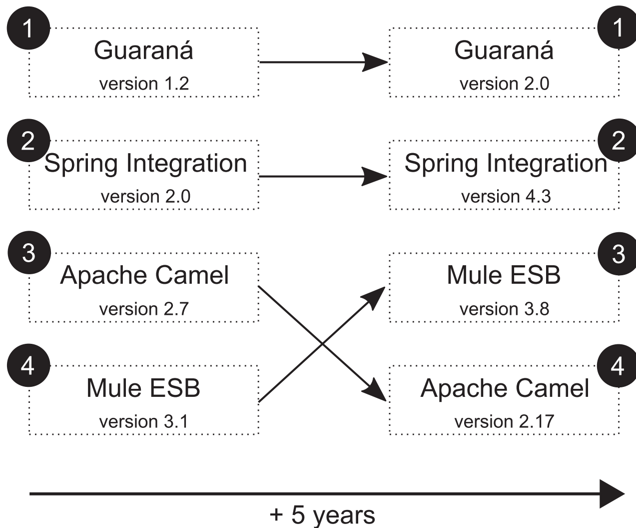

FIGURE 3 Comparing the empirical rankings of past and current versions analyzed

This change in the ranking is motivated by an increase in the total value of Apache Camel from 2.64 in its version 2.7 to 3.76 in its version 2.17, which was launched five years later.

7 | CONCLUSION

An enterprise usually uses several applications, either local or in the cloud. These applications comprise the enterprise software ecosystem and provide support for running business processes. The diversity of technologies, data models, and programming languages makes the exchange of data and the reuse of functionality from one application to another difficult, especially because the applications that comprise the software ecosystem were not often designed with integration in mind. Business processes are constantly being created and updated, frequently requiring collaboration between two or more applications. An integration solution aims to enable two or more applications to collaborate by exchanging data and sharing functionality. Integration frameworks are specialized software tools, which are built and adapted to provide support for the design and implementation of integration solutions. There are several open-source integration frameworks available on the market, which are designed to operate in a business context to manipulate structured data; however, increasingly, they are required to deal with unstructured and large volumes of data, thus requiring effort to adapt these frameworks to work with unstructured and large volumes of data. Choosing the framework, which is the easiest to be adapted, is not a trivial task.

In this article, we have analyzed the current stable versions of Apache Camel, Spring Integration, Mule ESB, and Guaraná, all of which are open-source integration frameworks. We followed a methodology to compute their maintainability metrics and rank them according to maintainability. The data obtained for the metrics in the last version of each framework were compared with a previous study, which allowed us to evaluate the analyzed integration frameworks during the five years of their evolution with respect to their maintenance. In this article, our sole purpose was to analyze the maintainability of these integration frameworks and to compare them; we did not intend to present a decision method with which to choose the best integration framework because this decision involves analyzing a variety of other factors, in addition to maintainability, such as documentation, training courses, technical support, and the set of adapters, related to the integration framework.

It is noticeable that every framework grew in size, and we saw the number of differences reflected in the number of lines of code, the number of packages, the number of classes, the number of interfaces, and the number of methods and attributes. Consequently, other metrics were also affected. Regarding coupling, some metrics, such as coupling intensity, increased in every framework, which indicates that the methods in every framework were more dependent on other methods. This is not necessarily negative because it could represent an improvement in method reuse. McCabe cyclomatic complexity did not decrease to a recommended level in Apache Camel, Spring Integration, or Mule ESB. In the best cases, it remained at the same level; however, in the worst cases, it was higher across the versions. Even if Guaraná increased its complexity, it was near to the recommended value. Our study shows that the development of newer versions of these frameworks is not concerned enough with improving maintainability; thus, more time may be required to understand the

functionality of a framework and adapt it to a specific context. Inheritance metrics, such as the depth of the inheritance tree and the number of overridden methods, had similar results concerning the maximum values for all frameworks, which changed proportionally to their size. For the depth of inheritance, Guaraná was the integration framework with the smallest value, in turn, implicating its compact size, making it easier to understand its functionality and be adapted to a specific context. Regarding the number of children classes of a class, Apache Camel had the highest value, followed by Mule ESB and Spring Integration with 812, 588, and 229, respectively. Guaraná was the framework with the smallest value for this metric, that is, 15. It is possible to observe that the metrics in Spring Integration increased more, in terms of percentage when considering the total values, than in the other integration frameworks. This result can be credited to the rapid growth of this integration framework, moving quickly from versions 2.0 to 4.3.

The new empirical ranks computed by ourselves in this review of the current versions of Apache Camel, Spring Integration, Mule ESB, and Guaraná reflect how these frameworks have evolved over five years. These ranks differ from the original ones, which reflects how each framework has become more concerned about maintenance. In the new empirical ranking, Mule ESB moved up one position to third place, whereas Apache Camel dropped one position. Bergmann-Hommel's test for the total values regarding each pair of proposals ranks the four integration frameworks at the same level. However, when analyzing the mean values computed for the metrics, Bergmann-Hommel's test ranks Guaraná in first place, and Spring Integration, Mule ESB, and Apache Camel in equal second place. In the future, we plan to include in our study other open-source integration frameworks that were not considered herein and in the original article, which proposed the methodology. We also plan to compute the maintainability metrics for all intermediate versions during this period of five years to track the evolution of individual metrics across the versions and possibly identify correlations between project improvement decisions and their actual reflection on source code maintainability.

ACKNOWLEDGEMENTS

This work was supported by the Brazilian Co-ordination Board for the Improvement of University Personnel (CAPES) under Grant 88881.119518/2016-01, and by the Research Support Foundation of the State of Rio Grande do Sul (FAPERGS) under Grant 17/2551-0001206-2. We would like to thank Ms. Elizabeth Thornton Rush from the Pennsylvania State University (United States) for her helpful comments in earlier versions of this article.

ORCID

Rafael Z. Frantz https://orcid.org/0000-0003-3740-7560

REFERENCES

1. Manikas K. Revisiting software ecosystems research: a longitudinal literature study. *J Syst Softw*. 2016;117:84-103.
2. Bosch J, Bosch-Sijtsema P. From integration to composition: on the impact of software product lines, global development and ecosystems. *J Syst Softw*. 2010;83(1):67-76.
3. dos Santos RP, Werner CML. A proposal for software ecosystems engineering. In: Proceedings of the Workshop on Software Ecosystems; 2011; Brussels, Belgium.
4. Linthicum DS. *Enterprise Application Integration*. Boston, MA: Addison Wesley; 1999.
5. He W, Xu LD. Integration of distributed enterprise applications: a survey. *IEEE Trans Ind Inform*. 2014;10(1):35-42.
6. Freire DL, Frantz RZ, Roos-Frantz F, Sawicki S. Survey on the run-time systems of enterprise application integration platforms focusing on performance. *Softw Pract Exp*. 2019;49(3):341-360.
7. Freire DL, Frantz RZ, Roos-Frantz F. Ranking enterprise application integration platforms from a performance perspective: an experience report. *Softw Pract Exp*. 2019;49(5):921-941.
8. Zimmermann O, Pautasso C, Hohpe G, Woolf B. A decade of enterprise integration patterns: a conversation with the authors. *IEEE Softw*. 2016;33(1):13-19.
9. Ritter D, May N, Rinderle-Ma S. Patterns for emerging application integration scenarios: a survey. *Inf Syst*. 2017;67(Supplement C):36-57.
10. Alexander C, Ishikawa S, Silvertein M. *A Pattern Language: Towns, Buildings, Construction*. Oxford, UK: Oxford University Press; 1977.
11. Hohpe G, Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA: Addison-Wesley Professional; 2004.
12. Ibsen C, Anstey J. *Camel in Action*. Greenwich, CT: Manning Publications Co.; 2010.
13. Fisher M, Partner J, Bogoevice M, Fuld I. *Spring Integration in Action*. Greenwich, CT: Manning Publications Co.; 2012.
14. Dossot D, d'Emic J, Romero V. *Mule in Action*. Greenwich, CT: Manning Publications Co.; 2014.
15. Frantz RZ, Corchuelo R, Roos-Frantz F. On the design of a maintainable software development kit to implement integration solutions. *J Syst Softw*. 2016;111(1):89-104.

16. da Silva RF, Filgueira R, Pietri I, Jiang M, Sakellariou R, Deelman E. A characterization of workflow management systems for extreme-scale applications. *Futur Gener Comput Syst*. 2017;75:228-238.
17. Linthicum DS. Cloud computing changes data integration forever: what's needed right now. *IEEE Cloud Comput*. 2017;4:50-53.
18. Ebert Nico, Weber Kristin, Koruna Stefan. Integration platform as a service. *Bus Inf Syst Eng*. 2017;59(5):375-379.
19. Guttridge K, Pezzini M, Golluscio E, Thoo E, Iijima K, Wilcox M. *Magic Quadrant for Enterprise Integration Platform as a Service 2017*. Stamford, CT: Gartner, Inc; 2017.
20. Ritter D., Rinderle-Ma S. Toward application integration with multimedia data. Paper presented at: IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC); 2017; Quebec City, Canada.
21. Radatz J, Geraci A, Katki F. *IEEE Standard Glossary of Software Engineering Terminology*. Piscataway, NJ: IEEE; 1990.
22. Lanza M, Marinescu R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. New York, NY: Springer Science & Business Media; 2007.
23. Lajios G. Software metrics suites for project landscapes. Paper presented at: 13th European Conference on Software Maintenance and Reengineering; 2009; Kaiserslautern, Germany.
24. Herraiz I, Izquierdo-Cortazar D, Rivas-Hernández F. Flossmetrics: Free/libre/open source software metrics. Paper presented at: 13th European Conference on Software Maintenance and Reengineering; 2009; Kaiserslautern, Germany.
25. Risi M, Scanniello G, Tortora G. Metric attitude. Paper presented at: 17th European Conference on Software Maintenance and Reengineering; 2013; Genova, Italy.
26. Li W, Henry S. Object-oriented metrics that predict maintainability. *J Syst Softw*. 1993;23(2):111-122.
27. Sheldon FT, Jerath K, Chung H. Metrics for maintainability of class inheritance hierarchies. *J Softw Maint Evol Res Pract*. 2002; 14(3):147-160.
28. Genero M, Piattini M, Calero C. A survey of metrics for UML class diagrams. *J Object Technol*. 2005;4(9):59-92.
29. Mouchawrab S, Briand LC, Labiche Y. A measurement framework for object-oriented software testability. *Inf Softw Technol*. 2005;47(15):979-997.
30. Briand LC, Daly JW, Wuest J. A unified framework for cohesion measurement in object-oriented systems. In: Proceedings of the 4th International Symposium on Software Metrics; 1997; Washington, DC.
31. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Trans Softw Eng*. 1994;20(6):476-493.
32. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ: Prentice-Hall, Inc.; 1996.
33. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall; 2002.
34. McCabe TJ. A complexity measure. *IEEE Trans Softw Eng*. 1976;2(4):308-320.
35. Frantz RZ, Corchuelo R, Roos-Frantz F. A methodology to evaluate the maintainability of enterprise application integration frameworks. *Int J Web Eng Technol*. 2015;10(4):334-354.
36. Fowler M. *Domain-Specific Languages*. Boston, MA: Addison-Wesley; 2010.
37. Metrics 1.3.6. Accessed June 15, 2018. 2005.
38. Dong X, Godfrey MW. Understanding source package organization using the hybrid model. Paper presented at: International conference on software maintenance; 2009; Edmonton, Canada.
39. Lorenz M, Kidd J. *Object-Oriented Software Metrics: A Practical Guide*. Upper Saddle River, NJ: Prentice-Hall, Inc.; 1994.
40. Offutt J, Abdurazik A, Schach SR. Quantitatively measuring object-oriented couplings. *Softw Qual J*. 2008;16(4):489-512.
41. Yu L. Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin. *J Braz Comput Soc*. 2008;14:45-55.
42. Marinescu R. *Measurement and Quality in Object-Oriented Design* [PhD thesis]. Timisoara, Romania: Department of Computer Science, Polytechnic University of Timisoara; 2002.
43. Alkadi Ghassan, Alkadi Ihssan. Application of a revised DIT metric to redesign an OO design. *J Object Technol*. 2003;2(3):127-134.
44. Demšar Janez. Statistical comparisons of classifiers over multiple data sets. *J Mach Learn Res*. 2006;7(1):1-30.

How to cite this article: Frantz RZ, Rehbein MH, Berlezi R, Roos-Frantz F. Ranking open source application integration frameworks based on maintainability metrics: A review of five-year evolution. *Softw: Pract Exper*. 2019;1–19. https://doi.org/10.1002/spe.2733