Extraction of Architectural Patterns from Frameworks and Modeling their Contributions to Qualities

HIND MILHEM, University of Ottawa
MICHAEL WEISS, Carleton University
STÉPHANE S. SOMÉ, University of Ottawa

Context: A framework is a highly reusable design for an application or part of an application in a given domain. It often defines the basic architecture of the applications that use it. With the increasing complexity of developing modern software systems and shorter delivery times, it is essential to reuse existing designs in the form of frameworks as much as possible. Therefore, selecting frameworks and documenting the underlying design rationale becomes an important task for system architects. An architect always needs to justify his/her architecture decisions, particularly when it comes to choosing among multiple frameworks. **Problem**: Selecting frameworks has been done in the past based on different characteristics and criteria. There were no studies identify the key criteria and characteristics of frameworks to enable a more informed choice by architects. **Objective**: In this paper, we try to provide a way to compare automatically the framework quality attributes based on specific characteristics of a framework. **Method:** In this paper, we extract the implemented architectural patterns from a framework's source code and document them to connect frameworks to quality requirements upon which a selection can be made. We use a tool called Archie (a tool used to extract tactics from code and produces a candidate set of tactics of a Java-based system) to extract the implemented architectural patterns of frameworks. We then document and model the patterns implemented by a framework and their impact on quality attributes using the Goal-oriented Requirements Language (GRL). **Results:** The satisfaction level of the quality requirements by a set of architectural patterns of a framework, integrated with other criteria and realistic design decisions context provide architects with a tool for comparing different frameworks and documenting their rationale for choosing a framework. To illustrate and as an initial validation of the approach, we apply it to a simple but realistic case study (choosing a stream processing framework for a cyber fusion center) with promising results.

Categories and Subject Descriptors: • **Software and its engineering→Software architectures** • **Software and its engineering→Patterns** • **Software and its engineering→Frameworks** • Software and its engineering→Extra-functional properties

General Terms: Framework Selection

Additional Key Words and Phrases: Architectural pattern, Non-Functional Requirement (NFR), data streaming framework, pattern extraction, framework modeling

1. INTRODUCTION

A framework is a highly reusable design for an application or part of an application in a given domain. It often defines the basic architecture of the applications that use it. With the increasing complexity developing software systems, it is essential to reuse existing designs in the form of frameworks. Selecting frameworks and documenting the underlying design rationale becomes an important task for system architects. An architect always needs to justify his/her architecture decisions, particularly when it comes to choosing among multiple frameworks.

Comparisons of frameworks have been done in the past to select the best framework based on different characteristics and criteria (Cervantes et al. 2013)(Grau and Xavier 2007)(Zalewski 2013). However, none of the past comparisons provide a way to compare automatically the framework quality attributes. Architectural Patterns are solutions that describe specific problems and their contexts (Bass et al. 2012). They define the basic characteristics and behavior of a system. Knowing the characteristics, strengths, and weaknesses of each architecture pattern are necessary in order to determine whether they push or pull a system toward or away from a quality requirement. The use of architectural patterns is considered a good practice to design software architectures with quality, improving the reuse and understanding of the rationale of architectural decisions. In the context of the data streaming frameworks, architectural patterns are used to connect these frameworks to quality requirements upon which a framework selection can be made. Using patterns encourage consistency and increase the speed of development. Patterns also help users of a framework to understand and leverage the framework (Carey and Carlson 2002). Unfortunately, the patterns used during the design of a framework are often not documented 0). Previous studies have introduced the idea of using patterns to document frameworks 0)(Aguiar, and David 2011).

Our research builds on this prior work and uses the documentation of the patterns that are implemented in a framework to choose among frameworks. In this paper, we extract the implemented architectural patterns from frameworks source codes to connect frameworks to quality requirements upon which a selection can be made. The framework selection is based on the satisfaction levels of given Non-Functional Requirements (NFRs) (Bass et al .2012). The rationale about the satisfaction levels of given NFRs for a framework would be

integrated with other criteria and realistic design decisions context to help an architect choose among multiple candidate frameworks. These criteria can be cost, maturity, stability, or community of a framework. In this paper, we use a tool called Archie (a tool used to extract a set of candidate tactics of a Java-based system) (Mirakhorli 2014) (Mirakhorli and Cleland-Huang 2016) (Mirakhorli et al. 2014) to find the patterns implemented in frameworks. Since Archie is developed to extract tactics (design decisions address specific quality attributes), we are also looking for extracting tactics in addition to patterns. However, in this paper, we only focus on patterns. We then model the patterns used by a framework and their impact on quality attributes using the Goal-oriented Requirements Language (GRL) (Mussbacher et al. 2007). The satisfaction levels of NFRs and the tradeoffs analysis of given NFRs, integrated with other criteria and realistic design decisions context can be used to help an architect select the best framework.

The remainder of this paper is organized as follows, Section 2 presents the research questions of this paper. Section 3 defined the important concepts considered in this work. Section 4 provides an overview of the related work. We proposed our approach in Section 5. Section 6 presents a case study illustrating the problem. Section 7 provides the analysis of the results to answer the research questions. In Section 8, we provide a discussion about general applied approach. Section 9 presents the threats to validity of this work. Section 10 draws initial conclusions and describes plans for future work.

2. RESEARCH QUESTIONS

We formulated a primary research question and two secondary sub-questions as shown in Table 1.

Table 1 Primary and Secondary Questions

PRIMARY REASEARCH QUESTION
Can the patterns be used as a good criterion and characteristic of candidate frameworks to enable a more informed choice by architects?
SECONDAY SUB-QUESTIONS
1- Is it possible to determine (extract) the patterns used by a framework?
2- Is it possible to compare candidate frameworks based on their implemented patterns and select the best one for a given set of quality attributes?

3. BACKGROUND

In this section we present important concepts considered in this work. We presented the definition of architectural patterns, Non-Functional Requirements (NFRs), and data streaming frameworks.

3.1 Architectural Patterns

Architectural Patterns are solutions that describe specific problems and their contexts. Examples of architectural patterns are the Broker pattern (Buschmann et al. 1996), Layers pattern (Buschmann et al. 1996), and Pipes and Filters pattern (Buschmann et al. 1996). Architectural patterns express high-level design decisions and describe high-level structures and behaviors of systems (Bass et al. 2012)(Rozanski, and Woods 2012). The architectural patterns can satisfy the functional requirements, non-functional requirements (NFRs), and constraints of a system. An architectural pattern may have positive or negative impacts on specific NFRs.

3.2 Architectural Tactics

Architectural tactics are design decisions that affect the achievement of NFRs response and are used to address the quality requirements (Bass et al. 2012). Such as Heartbeat (Bass et al. 2012), Ping/Echo (Bass et al. 2012), Authentication (Bass et al. 2012), and Authorization (Bass et al. 2012). While architectural patterns express high-level design decisions, an architectural tactic is a design strategy that addresses a particular NFR (Bass et al. 2012)(Rozanski, and Woods 2012). In general, the tactic has structure and behavior and can influence architectural patterns in several ways. In one way, the tactics can be implemented in the same structure of architectural patterns. In another way, the tactics may require changes to the structure and behavior of architectural patterns.

3.3 Non-Functional Requirements (NFRs)

Non-functional requirements or what is called (Quality Attributes (QAs) or quality requirements) are characteristics that the system has. They are qualifications of the functional requirements or the overall product. NFRs include performance, security, usability, and reliability (Bass et al. 2012)(Rozanski, and Woods 2012). These qualifications should be considered with the functions of the system. No system's function can stand without due consideration of other NFRs. For example, if a system has as function that the pressing of a green button should make an options dialog to appear. Then, a NFR performance might describe how quickly that dialog should appear. A NFR availability also might express how often this function should fail, and so on. The satisfaction of NFRs is generally not a simple binary determination (i.e. met or not). The level of satisfaction is generally over a specific scale which considers specific contexts and scenarios. The specification of an NFR includes several parts (Bass et al. 2012): Source of stimulus (some entities such as human, computer), Stimulus (condition requires response), Environment (certain conditions the stimulus occurred under them), Artifacts (the whole system, part of system, or collections of systems), Response (activity will be taken when the stimulus arrives), and Response Measure (measures the response when it occurs).

3.4 Data Streaming Frameworks

Data stream processing has recently became very important due to the steadily growing number of data sources that continuously produce and offer data. Data streaming frameworks such as Apache Metron[1], Apache Flink[2], Apache Spark[3], Apache Spot[4], and Apache Storm[5] are mainly designed to process huge amount of data streams and to make decisions. Various organizations have started to employ such data streaming frameworks to solve major emerging big data problems related to smart ecosystems, healthcare services, social media, etc [Inoubli et al. 2018]. For example, Social media is a representative data source for big data that requires real-time processing and results [Vlassopoulos et al. 2016].

4. RELATED WORK

There are some alternative methods to Archie (Mirakhorli 2014) (Mirakhorli and Cleland-Huang 2016) (Mirakhorli et al. 2014) can be used to extract the patterns of a framework. These methods are discussed in the following related work.

Cervantes et al. (Cervantes et al. 2013) extract patterns and tactics from a framework by applying a mapping process between the patterns and tactics in a framework and those patterns and tactics which are employed in architecture design. They also mention that patterns can be extracted from the provided services of a framework and that framework selection is based on architecture drivers (such as the team's level of knowledge of a framework, or the framework's maturity).

Beck and Johnson (Beck and Johnson 1994) extract patterns from the problem statement of an architecture to document the HotDraw framework. They describe the problem statements of HotDraw framework and then recognize the patterns which were used to solve these problem statements.

Ryoo et al. (Ryoo et al. 2015) perform interviews with an architect to get the employed security tactics in architectures. Then, they describe the architectures in terms of these collected tactics. They focus only on tactics. However, interviews can be also used to extract the patterns from an architecture.

Meusel et al. (Meusel et al. 1997) describe the process of patterns instantiation by assigning roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a concrete design. This method has been defined to extract the patterns from a design.

Sena et al. (Sena 2018) analyze studies reporting on software architectures of big data systems, to identify architectural patterns, quality attributes, as well as problems and liabilities of those patterns. They determined that various architectural patterns, such as the Layered pattern, the Pipe and Filter pattern, the Broker pattern, and the Shared Repository pattern have significant impacts on the qualities and characteristics of big data systems.

[1]Apache Metron, metron.apache.org

[2]Apache Flink, flink.apache.org

[3]https://spark.apache.org

[4]https://spot.incubator.apache.org

[5]https://storm.apache.org

5. PROPOSED APPROACH

We propose an approach that identifies the patterns which are implemented by a framework, and then model the framework in terms of these implemented patterns to connect frameworks to quality requirements upon which a selection can be made. The approach consists of two main steps: A) determining patterns implemented in a framework, B) modeling the frameworks in terms of their implemented patterns.

5.1 Determining patterns implemented in a framework

We use Archie (Mirakhorli 2014) (Mirakhorli and Cleland-Huang 2016) (Mirakhorli et al. 2014) to extract patterns from the frameworks source code as discussed in Mirakhorli's and Huang's work in (Mirakhorli and Cleland-Huang 2016). We chose Archie because it is the only automated tool compared with the other methods, which are mentioned in the related work. It is a plugin of the Eclipse platform. It is also an extensible tool so we can add or remove patterns. It has an interactive interface so we can run more than one framework.

Mirakhorli and Huang trained a classifier in Archie to recognize specific terms that occur commonly across implemented tactics, and calculate the weights of the tactics (the probability that a particular term identifies a class associated with a tactic). Archie tool considers thirteen tactics (Mirakhorli 2014) (Mirakhorli and Cleland-Huang 2016) (Mirakhorli et al. 2014) from three quality attributes to be detected in any Java-based system. These tactics are Policy-Based Access Control (PBAC), Role Based Access Control (RBAC), Kerberos, Audit trail, Session Management, and Authenticate from Security, Checkpoint, Heartbeat, Ping/Echo, Active Redundancy, and Load Balancing form Reliability, and Resource Scheduling, and Resource Pooling from Performance. In addition to these thirteen tactics, we added other tactics and patterns to be detected by Archie tool for a data streaming framework as we will see in Section 5.1.1. We determined the patterns based on the same information retrieval approach used by Mirakhorli as discussed in Section 5.1.2. As a result of applying the Archie tool, a set of patterns can be detected and presented to an architect as potential patterns of a framework.

5.1.1 Choosing the patterns that need to be checked in a data streaming framework

To choose the patterns that need to be checked in a data streaming framework, we conducted a literature review to find the most relevant patterns of a big data system in general and a data streaming system in specific. Sena et al. [18] conducted a systematic mapping study to analyze studies reporting software architectures of big data systems. They identified a set of requirements and modules for the big data systems. Sena et al. (Sena et al. 2018) used the studies which identified in their previous work in (Sena et al. 2017) to identify the common used architectural patterns for big data systems. The patterns which are determined by Sena et al. in (Sena et al. 2018) are "the Layers pattern," "the Pipes and Filters pattern," "the Broker pattern," and "Shared-Repository." We performed a literature search for publications reporting on patterns for big data systems from the year of 2017 until present, to supplement Sena's list (as this work was completed in 2017). As we can see in Table 6 in the Appendix, we found sixteen primary studies addressing patterns. We got the first six studies using the same search string as Sena's (Sena et al. 2017)(Sena et al. 2018). S7 and S8 were obtained from the references of S2. We got the studies S9-S16 using our search string "(("Reference Architecture" OR "Reference Model") AND "Data Streaming System")."

We filtered those patterns which are shown in Table 7 such that we only get the list of the commonly used patterns for data streaming frameworks as shown in Table 2. This has been done by looking just to the studies which reported to use of patterns in data streaming frameworks. During our review, we also searched about the NFRs or quality attributes which are addressed in big data systems in general as shown in Table 8 in the Appendix. We filtered those NFRs which are shown in Table 8 such that we only get the list of the commonly used NFRs for data streaming frameworks as shown in Table 3.

Table 2 The Most Commonly Used Patterns in Data Streaming Systems

PATTERNS
Layers
Broker
Pipes and Filters
Shared-Repository
Observer/Publish-Subscribe

Table 3 The Most Commonly Used NFRs in Data Streaming Systems

NON-FUNCTIONAL REQUIREMENTS (NFRS)	
Scalability	Security
Maintainability	Portability
Performance	Interoperability
Reliability	Availability

5.1.2 Adding patterns to the Archie

In this sub-step, we modified Mirakhorli's approach to support patterns by adding to the Archie all the patterns which are shown in Table 2 to be detected by Archie tool. Then, we added terms related to the patterns. In Mirakhorl's approach, a classifier in Archie is trained to recognize specific terms that occur commonly across implemented tactics, and calculate the weights of the tactics (the probability that a particular term identifies a class associated with a tactic). Compared to Mirakhorli's approach, we could recognize these terms from the description of the patterns such as the name, the problem, the context, and the solution of a pattern. We looked at different descriptions from different sources (Grau, and Franch 2007) (Siu and Yu 1995) (ISO/IEC 19505 2012)(OMG 2012)(Tyree and Akerman 2005)(Bass et al. 2012)(Buschmann et al. 2007)(Buschmann et al. 1996) to get the descriptions of the patterns. Then, we extract from the descriptions only the most frequent and related terms of a pattern. We ignored the non-related words such as the 'stop' words, the conjunction words, etc. Additional related terms are determined during our manual search in the source code where a pattern is implemented. Then, we added these terms to the Archie so its detector can search to retrieve the patterns in the source code of a framework. Archie search about the related terms of a pattern everywhere in the source code (i.e. in the method names/parameters, variables names, classes names, and the comments. This is because numerous studies (Antoniol et al. 2002)(Cleland-Huang et al. 2007) have shown that developers tend to use meaningful terms to name variables, methods, and classes. The studies have shown also that developers often provide meaningful comments which offer insights into the purpose of the code (Mirakhorli and Cleland-Huang 2016). This would validate that our added terms are the good ones especially that our search covered different descriptions of a pattern from different sources. For example, in the description of the Broker pattern as documented in (Buschmann et al. 1996) as shown in the Appendix, we highlighted the most common used, the most frequent, and related terms in the description. As we can see, the most frequent and related terms to the Broker pattern are *broker, client, server, distributed,* and *system*. We could also extract additional terms from different descriptions of the Broker pattern such as the one that is documented in (Sena et al. 2018). The additional terms are *router, intermediary, provider, producer, consumer,* and *transformer*. Note that by following Mirakhorli's approach, we only consider single terms not pairs or triplets of terms.

The related terms of the Broker pattern together are *broker, distributed, system, client, server, router, intermediary, provider, producer, consumer,* and *transformer*.

Our added patterns with their related terms are shown in Table 4. As a result of applying the Archie tool on the source code of a framework, a set of patterns can be detected and presented to an architect as candidate patterns of a framework.

Table 4 Added patterns and their related terms

ADDED PATTERNS	ADDED RELATED TERMS
Layers (Buschmann et al. 1996)	layer, tier, responsibility, functionality
Observer/Publish-Subscribe (Buschmann et al. 1996)	Observe, publish, subscribe, listen
Pipes and Filters pattern (Buschmann et al. 1996)	Pipe, filter, sequence
Broker (Buschmann et al. 1996)	broker, client, server, distributed system, consumer, producer, intermediary, router, provider, transformer
Shared Repository (Buschmann et al. 1996)	reposit, repository, store, storage

5.2 Modeling frameworks in terms of their implemented patterns

In this step, we model a framework in terms of the patterns it implements to satisfy given NFRs. Before we model a framework, we 1) Determe the NFRs and the contributions of the detected patterns on the NFRs. 2) Model the patterns and their contributions on the NFRs.

After determining and modeling the patterns and their contributions on the NFRs, we model the frameworks in terms of their implemented patterns. In the following, we discuss the two sub-steps.

5.2.1 Determining NFRs and the contributions of the detected patterns on the NFRs

We could extract the NFRs, the contributions of the patterns on the NFRs, and the design decisions, which show the reason of the negative or the positive impact of a pattern on an NFR, from the documentation of the patterns. We consider the descriptions of the patterns as documented in (Buschmann et al. 1996) and (Buschmann et al. 2007) but only focus on the consequences and solution sections.

We follow Ong et al.'s (Ong et al. 2003) approach to extract NFRs, design decisions, and the contributions of the patterns on the NFRs. Such that, we added to the description by underlining the benefits, liabilities, the affected NFRs, and reasons for the positive or negative impact of the patterns on the NFRs. The benefits and liabilities of a pattern indicate the positive and negative contributions on the NFRs respectively. The reasons for the positive or negative impact of the patterns on the NFRs indicate the design decisions behind a pattern. These design decisions are expressed as phrases starting with an active verb such as define, register, change, reuse, etc. We present the documentation of the Broker pattern in the Appendix as documented in (Buschmann et al. 1996) as an example.

5.2.2 Modeling the patterns and their contributions on the NFRs

We derive GRL models with the NFRs and the contributions of the patterns on the NFRs from the description of each pattern. . First, we start with the patterns at the bottom of the model. Then, we put the design decisions and NFRs at the topmost level of the model. The complexity of the system dictates the number of levels of design decisions as shown in Figure 1.

Based on Figure 2, we select softgoals (clouds) elements to represent NFRs and the design decisions, indicating that these cannot be achieved in an absolute manner. Tasks (hexagons) are selected to represent patterns and frameworks, representing ways of achieving a softgoal. Resources (rectangles) are selected to represent the components or the parts of the framework where the patterns are implemented. Solid lines (Contribution links) indicate the desired impacts of one element on another element. Contribution types determined by labels. These labels indicate various degrees of positive (+) or negative (-) contributions (see Figure 2 for the complete set of labels). Decomposition links allow an element to be decomposed into sub-elements (Mussbacher et al. 2007). AND, IOR and XOR are supported decompositions. We use only AND decomposition links to represent the connection between a framework and its patterns, because all the patterns are required in a framework before the NFRs are satisfied. We used it also to represent the connection between the parts of a framework and the patterns, because all the patterns are needed to be implemented in a part of a framework. The satisfaction levels Denied, Weakly Denied, Satisfied, Weakly Satisfied, Conflict, Unknown, None, and Exceeds are used to represent the satisfaction level of NFRs.

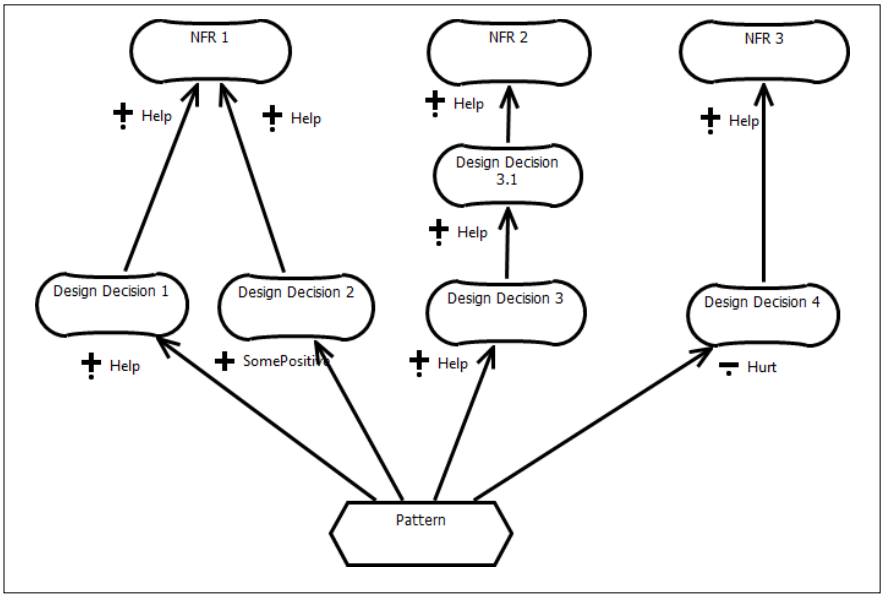

Fig. 1. The general model of a pattern and its contributions on NFRs

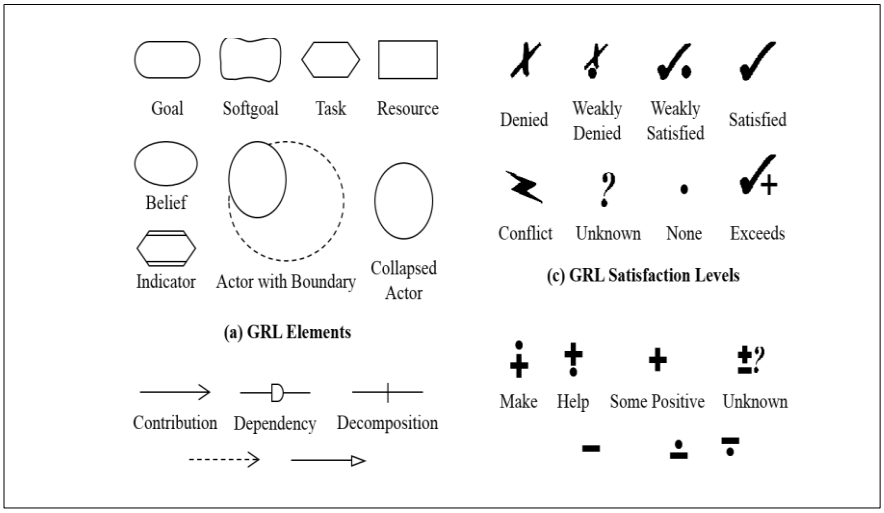

Fig. 2. Summary of the GRL notation (Mussbacher et al. 2007)

The general GRL model of a pattern is shown in Figure 1. As we can see, the NFRs being at the top of the design decisions. The reason for this arrangement is that the design decisions push or pull the framework towards or away from NFRs (e.g., better security or greater availability). As design decisions connect patterns and NFRs, they provide an explanation of why a pattern impacts an NFR the way it does.

We derived the GRL model of the Broker pattern and its contribution to the NFRs from its documentation, which is shown in the Appendix, as shown in Figure 3. For example, the Broker pattern helps the low coupling services by using direction layers such as APIs, bridges, and proxies to hide the operating system and network system details from clients and servers. This improves Portability. Therefore, a positive contribution (Help) is shown between "Broker" and "Portability." However, failing of the broker during program execution would cause all the applications that depend on it to be unable to continue successfully. Therefore, a negative contribution (Hurt) is shown between "Broker" and "Reliability." Tested services make client applications more robust and easier to test but on the other hand, testing the whole broker is tedious because of many components. This improves and at the same time decreases the "Maintainability." Therefore, a positive contribution of Help is shown between "Broker" and “Maintainability," and at the same time a negative contribution of Hurt is shown between "Broker" and “Maintainability." This results in an unknown satisfaction level.

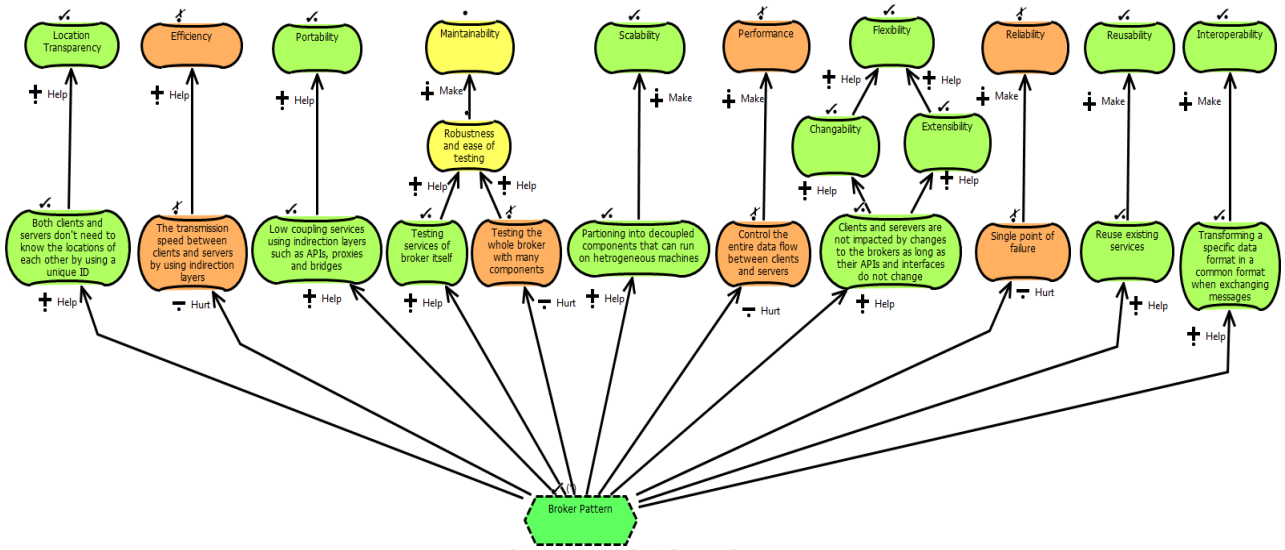

Fig. 3. The GRL model of the Broker pattern

We derived the GRL models for all the determined patterns in this work from their descriptions, following the general model in Figure 1.

These GRL models of the patterns, were then used to build a bottom-up GRL model for frameworks, starting with the framework and its parts at the bottom level of the model, connected with all its implemented patterns. The resulting GRL model specifies that the design decisions provide an explanation of why a pattern impacts an NFR the way it does. Consequently, the design decisions push or pull the framework towards or away from NFRs.

The general GRL model of a framework in terms of its used patterns is shown in Figure 4. As we can see in Figure 4, the design decisions push or pull the framework towards or away from NFRs (e.g., better security or greater availability). For example, "Design Decision 2" and "Design Decision 2.1" pull "Framework" away from "NFR 2". This is because "Pattern 2" of Part 2 negatively impacts "Design Decision 2", which hurts "Design Decision 2.1", resulting in reduce the satisfaction of the "NFR 2" of "Framework."

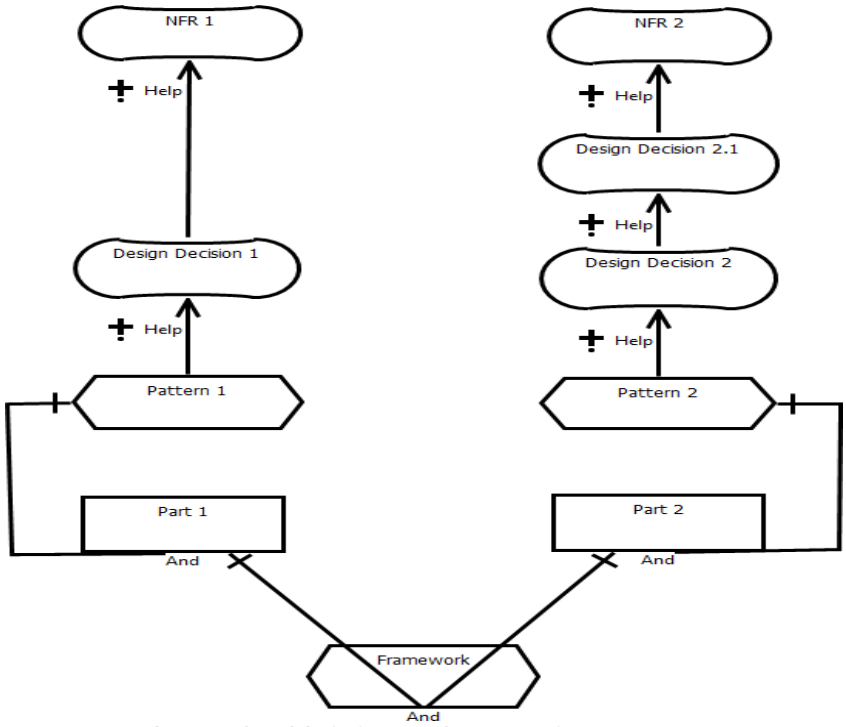

Fig. 4. The general model of a framework in terms of its patterns

6. CASE STUDY

As an initial validation of our approach, we present a simple but realistic case study of the task of choosing a stream processing framework for a cyber fusion center. The framework will provide the backbone for the collection and correlation of security events. Processing the events requires routing information from sensors to various processing stages that perform analytics on the events at different levels of abstraction (such as detecting attacks and attack patterns).

Two candidate frameworks were suggested for this project: Apache Strom (a component in Apache Metron) and Apache Flink. In the following, we applied the two main steps of our approach to extract the implemented patterns in the two frameworks.

6.1 Determining the patterns of Storm and Flink

We applied the Archie tool on the source code of Storm and Flink. Then, we validated the results of applying Archie on the Storm and Flink frameworks by hunting for the occurrences of those patterns, which are detected by Archie, manually in the source code/documentation/websites of Storm and Flink.

The analysis of the results of applying Archie on the Storm and Flink are shown in Tables 5 and 6 respectively. Tables 5 and 6 show the detected and non-detected patterns for Storm and Flink, the number of trained on terms (the terms that were used by Mirakhorli and our added ones). They also show the number of the detected terms (the terms that are detected by Archie in the source code of Storm and Flink), the number of Java classes where these terms are detected, and the number of Java classes covered by a pattern (the classes used in the implementation of a pattern). The tables also show if the pattern is detected by Archie or not, and the threshold where a pattern is detected (a selected number is given by Archie to determine the likelihood of that a given class is associated with a pattern). The tables also show the classification of the patterns as either True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN). Where TP means Archie detects a pattern and that is effectively implemented in Storm/Flink. While, TN means Archie does not detect a pattern, which is in fact not implemented in Storm/Flink. FP means Archie detects a pattern which is not implemented in Storm/Flink. While, FN means Archie does not detect a pattern, which is in fact, applied in Storm/Flink.

Table 5 The Analysis of the Results of Applying Archie on Storm

PATTERNS	NUMBER OF TRAINED ON TERMS	NUMBER OF DETECTED TERMS	NUMBER OF CLASSES WHERE TERMS ARE DETECTED	NUMBER OF CLASSES ARE COVERED IN A PATTERN	DETECTED BY ARCHIE (YES/NO)	TP/TN /FP/FN	THRESHOLD
Layers	4	1	4	4	Yes	TP	<=0.9
Broker	10	1	19	6	Yes	TP	<=1
Observer/Publish -Subscribe	4	1	3	3	Yes	TP	<=0.2
Pipes and Filters	3	1	123	15	Yes	TP	<=0.5
Shared-Repository	4	0	0	0	NO	TN	-
Total	**25**	**4**	**116**	**28**			

Table 6 The Analysis of the Results of Applying Archie on Flink

PATTERNS	NUMBER OF TRAINED ON TERMS	NUMBER OF DETECTED TERMS	NUMBER OF CLASSES WHERE TERMS ARE DETECTED	NUMBER OF CLASSES ARE COVERED IN A PATTERN	DETECTED BY ARCHIE (YES/NO)	TP/TN /FP/FN	THRESHOLD
Layers	4	1	24	5	Yes	TP	<=0.9
Broker	10	1	63	7	Yes	TP	<=1
Observer/Publish-Subscribe	4	2	61	8	Yes	TP	<=0.4
Pipes and Filters	3	2	254	22	Yes	TP	<=1
Shared-Repository	4	0	0	0	NO	TN	-
Total	**25**	**6**	**402**	**42**			

As we can see in Tables 5 and 6, both Storm and Flink frameworks has implemented the same patterns. Hence, there is no difference between the two frameworks Storm and Flink. This would confirm that Flink can be used as a component in Metron instead of Storm*. This case study does not show any differences between the two frameworks. In our future work, we will use another case study to show differences between the candidate frameworks.

6.2 Modeling the frameworks in terms of their patterns

In this step, we model the two candidate frameworks in terms of their detected patterns, following the general model in Figure 4 in Section 5.2.2. We only consider the NFRs which are relative to a data streaming system (the ones which are shown in Table 3). Hence, the final set of NFRs considered for Storm and Flink is Scalability, Maintainability, Performance, Portability, Availability, Reliability, Security, and Interoperability. For sake of readability, we separated each model into different models based on various aspects. However, we only define here one model for Storm and Flink from two aspects Reliability and Availability requirements as shown in Figure 5. As we can see in Figure 5, the design decisions provide an explanation of why a pattern impacts an NFR the way it does. Consequently, the design decisions push or pull the framework toward or away from NFRs. For example, applying the "Observer/Publish-Subscribe" pattern at the internal part pushes the Storm framework towards "Availability." This is justified by the fact that "Observer/Publish-Subscribe" pattern provides Asynchronous communication between components without blocking to wait for a response, which helps Decouples publisher and subscribers so they can be active and available at different points in time, resulting in improving of "Availability." The "Broker" pattern at the interface part introduces a "single point of failure" such that when the broker fails, the whole system stops working. This pulls the Storm framework away from the "Reliability" and the "Availability." It is also hard to handle the errors of a task/process by applying "Pipes and Filters" pattern at the libraries part of Flink because of the impossibility of for example restarting a pipeline or ignoring an error. This pulls the Flink framework away from the "Availability." Providing rational about how the implemented patterns in a framework can push or pull the framework toward or away from given NFRs, integrated with other criteria such as cost, delivery time, stability, maturity of a framework would help an architect choose among several candidate frameworks.

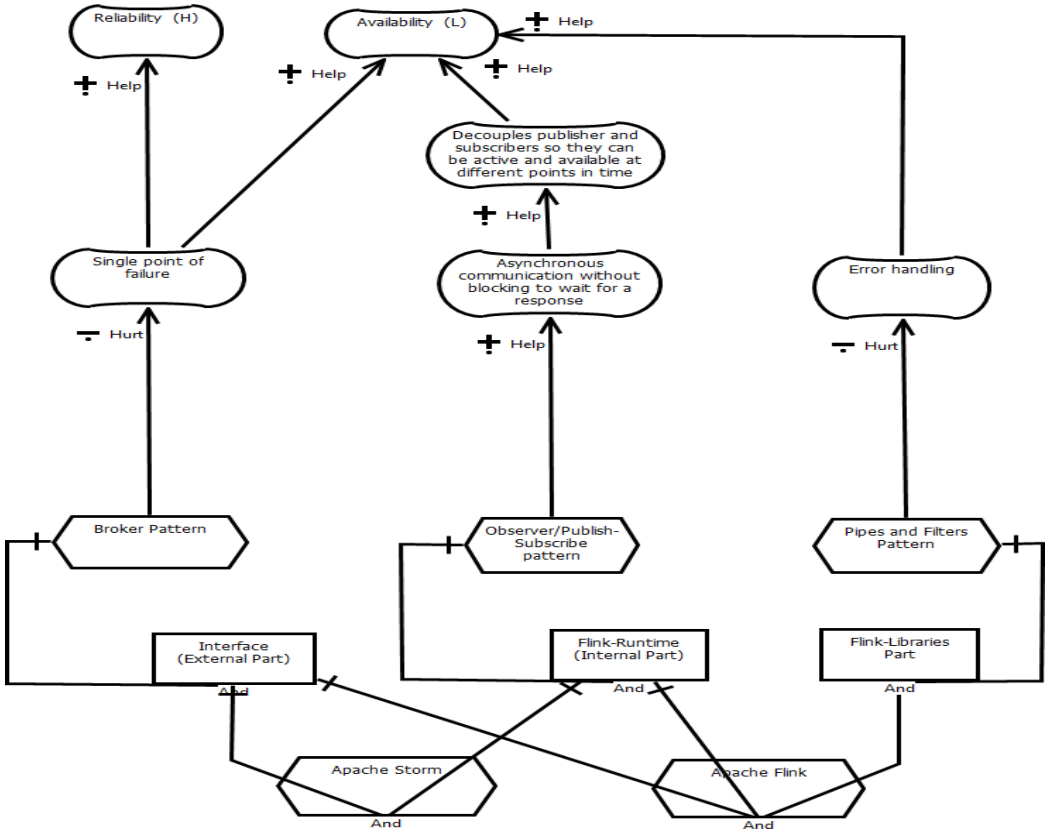

Fig. 5. The GRL model of Storm and Flink frameworks considering the reliability and availability requirements

7. ANALYSIS

In this section, we analyze the results of applying Archie on the case study so we can answer our research questions which are shown in Section 2.

7.1 Can the patterns be used as a good criterion and characteristic of candidate frameworks to enable a more informed choice by architects?

In this work, we present an approach to choose among multiple candidate frameworks. The approach uses the patterns extracted from the frameworks to compare and choose among different frameworks. We extract the patterns instantiated by the frameworks using an information retrieval approach. We used Archie to extract the patterns from the source code of the frameworks. A model of the frameworks in terms of their patterns was created to support the comparison and evaluate the quality attributes of the candidate frameworks.

The results of applying Archie on our case study shows that the implemented patterns for both Flink and Storm frameworks are the same. This would confirm that Flink can be used as a component in Metron instead of Storm*. Thus, analyzing the frameworks in terms of their implemented patterns provided us with a rational about which quality attributes are either improved or decreased by which patterns and in which part of the framework. Providing such rational integrated with other criteria such as cost, delivery time, stability, maturity of a framework would enable a more informed choice by architects. This would lead us to answer the primary question of this paper, which should be yes, analyzing candidate frameworks in terms of their implemented patterns is a good criterion to enable a more informed choice by architects

7.2 Is it possible to determine (extract) the patterns used by a framework?

The patterns which are used in frameworks, can be extracted using different methods and tools such as 1) Archie (Mirakhorli 2014) (Mirakhorli and Cleland-Huang 2016) (Mirakhorli et al. 2014) 2) matching method between the provided services of a framework and its patterns (Sena et al. 2018) 3) pattern instantiation (assigning the roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a practical design) (Aguiar, and David 2011), and 4) matching method between the problem statement of an architecture and the applied patterns (Beck and Johnson 1994). This would answer the first question of the secondary questions.

In this work, we used Archie to extract patterns from the frameworks source code. We chose Archie because it is the only automated tool compared with the other methods, which are mentioned in the related work. It is a plugin of the Eclipse platform. It is also an extensible tool so we can add or remove patterns. It has an interactive interface so we can run more than one framework.

7.3 Is it possible to compare candidate frameworks based on their implemented patterns and select the best one for a given set of quality attributes?

To answer this secondary questions, we created a model of the candidate frameworks in terms of their implemented patterns. The model support the evaluation of the quality attributes by providing the contribution values of the patterns on the quality attributes. This would provide a rational about which quality attributes are improved and which are not by which pattern(s) and in which part of a framework. Knowing the characteristics, strengths, and weaknesses of each architecture pattern are necessary in order to determine whether they push or pull a system toward or away from a quality requirement. Providing this rational with other criteria such as cost, delivery time, stability, maturity of a framework would enable a more informed choice by architects. In our case study, architectural patterns are used to connect both Flink and Storm frameworks to quality requirements upon which a framework selection can be made. There were no differences between Flink and Storm frameworks, which confirm that Flink can be used as a component in Metron instead of Storm*. Applying the approach on different case studies can help showing the differences between the frameworks to choose the best one.

* https://datahovel.com/2018/07/26/apache-metron-as-an-example-for-a-real-time-data-processing-pipeline/

8. DISCUSSION

In this paper, we determine the implemented architectural patterns of frameworks to be used as a criterion to connect frameworks to quality requirements upon which a selection can be made. We extract the patterns implemented in the frameworks using an information retrieval approach with a tool called Archie. In this paper, we used Apache Flink and Apache Storm as two candidate frameworks to our case study to apply the approach to.

In the following, we summarize the general steps of our approach in case of using different case studies, frameworks, and patterns:

1- Determine the context/domain of the project

By determining the context/domain of the case study, you will restrict the scope of your search. In our case study, because of the project we have (a cyber fusion center) concerns with processing data and events, which requires routing information from sensors to various processing stages, we determined the context of the project to be a data streaming context/domain

2- Determine the candidate frameworks based on the determined context/domain

Because of the Archie tool only works with Java-based systems, the selected candidate frameworks should be implemented using Java language. In our cases study, several data streaming frameworks were suggested for this project such as Apache Storm, Apache Metron, Apache Flink, and Apache Spot. However, two candidate frameworks were only suggested for our case study: Apache Strom (a component in Apache Metron) and Apache Flink. This is because that both Storm and FLink frameworks are Java-based systems.

3- Add the patterns to the selected tool (Archie)

We performed a literature review to determine the patterns that need to be checked for a data streaming framework. We then added those determined patterns to the Archie tool to be detected to a candidate framework.

4- Apply the tool on the candidate frameworks and get the patterns of each framework

We applied Archie on both candidate frameworks Storm and Flink and got set of candidate patterns for each framework. Then, we validated the results of applying Archie on the Storm and Flink frameworks by hunting for the occurrences of those patterns, which are detected by Archie, manually in the source code/documentation/websites of Storm and Flink.

5- Model the frameworks in terms of their implemented patterns

We model the two candidate frameworks in terms of their detected patterns, following the general model in Figure 4 in Section 5.2.2. We only consider the NFRs which are relative to a data streaming system (the ones which are shown in Table 2).

By following the above steps, you can apply the approach on any different Java-based systems (frameworks).

9. THREATS TO VALIDITY

Threats to validity can be classified as construct, internal, and external validity. We discuss the threats which potentially impacted our work, and the ways in which we attempted to mitigate them.

External Validity evaluates the generalizability of the approach. The primary threat is related to the identification of the related terms of a pattern in this work. The task of identifying these related terms was conducted manually by looking at different descriptions of a pattern from different sources, and checking the existence of the determined terms in the source code of the frameworks. The manual identification of the related terms in in different descriptions from different sources, and the existence of the related terms written by developers in the source codes gave us confidence that each of the identified term was indeed representative of its relevant pattern. However, it is more difficult to ensure that all related terms of a given pattern have been identified. For example, there could be terms that we failed to find. We mitigate this by considering different descriptions of a pattern as documented in different sources to identify the terms related to the pattern.

Construct Validity evaluates the degree to which Archie was accurate in detecting the patterns of the frameworks. In our case study, we have calculated the false positives and false negatives numbers. We found that there were no false positives and false negatives possibilities. The whole results showed that all the patterns were detected by Archie for both frameworks are in fact implemented in the frameworks. This confirms the high accuracy and the performance of the Archie tool. Archie also has been tested on several systems ranging from 1,000 to 20,000 java files. Hadoop is the case study which used to evaluate the Archie tool by calculating the number of false positives and false negatives. Hadoop is a large and realistic system has three major subsystems and many hundreds of programs. Therefore, evaluating Archie using a large and realistic system like Hadoop and calculating the false positives and false negatives numbers can confirm the high efficiency and the performance of the Archie tool and the possibility of generalize it to broader systems.

Internal Validity reflects the extent to which a work minimizes systematic error or bias, so that a causal conclusion can be drawn. A good threat to validity is that the search for specific patterns was limited by the literature of the Storm and Flink Frameworks, and that additional undiscovered patterns existed that used entirely different terminology. However we partially mitigated this risk through locating patterns using searching, browsing, and expert opinion. In the case of the cyber fusion center project, we personally elicited feedback from the developers and architects of the Storm and Flink frameworks.

10. CONCLUSION AND FUTURE WORK

The approach described in this paper extracts the implemented architectural patterns from frameworks source codes to connect frameworks to quality requirements upon which a selection can be made. We extract the patterns implemented in the frameworks using an information retrieval approach with a tool called Archie. We then model the frameworks in terms of their instantiated patterns using the Goal-oriented Requirements Language (GRL). This model provides architects with a rationale about the satisfaction levels and the tradeoffs analysis of given NFRs for a framework. This rationale can be integrated with other criteria and more realistic design decision context for choosing the best fit framework for given quality attributes. .

Initial results from applying the approach to a case study are promising, but more work is required to strengthen the approach. In future work, we plan to evaluate the GRL models and automatically calculate the satisfaction levels of given NFRs. We also plan to improve the modeling approach by first, considering the tactics to be included in the GRL model of a framework so the satisfaction level of NFRs can be improved; by secondly, calculating the overlap between the tactics and patterns; and thirdly, by assigning the satisfaction levels with reference to a corpus of open source projects.

APPENDIX

Table 7 Primary Studies Reporting Patterns in Big Data Systems

ID	TITLE	AUTHOR(S)/YEAR	PATTERNS	SEARCH STRING
S1	A big data analytics architecture for Industry 4.0	Santos et al. 2017	Layered	("Big Data" AND ("Software Architecture" OR "Reference Architecture" OR "Reference Model")) [**2017-Present**]
S2	Towards a Security Reference Architecture for Big Data	Moreno et al. 2018	----------	
S3	Investigating the Applicability of Architectural Patterns in Big data Systems	Sena et al. 2018	Layers, Broker, Pipes and Filters, Shared-Repository	
S4	A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability	Carrez et al. 2017	Broker, Data repository, Publish/Subscribe-observer,	
S5	Research on the Fusion Model Reference Architecture of Sensed Information of Human Body for Medical and Healthcare IoT	He et al. 2018	----------	
S6	Simplifying Big Data Analytics Systems with a Reference Architecture	Sang, et al. 2017	Publish/Subscribe, Broker.	

S7	BlueTalon Data-Centric Security Platform: Bringing Order to Data Security Chaos	Blue Talon, 2016	Layered	Reference of Moreno 2018
S8	Big Data and Data Centric Security	SQRRL, 2014	Layered	Reference of Moreno 2018
S9	A reference web architecture and patterns for real-time visual analytics on large streaming data	Kandogan et al. 2013	Federated Consumer, Observer, Repository, Blackboard	(("Reference Architecture" OR "Reference Model") AND" Data Streaming System") [**No restriction on the date**]
S10	Architectures for Streaming Data Processing in Sensor Networks	Kim et al. 2005	----------	
S11	On the Data Streaming Processing Frameworks: A Case Study	Dhaouadi et al. 2018	----------	
S12	A Reference Architecture for Big Data Systems	Sang, et al. 2016	----------	
S13	A survey on platforms for big data analytics	Singh et al. 2015	----------	
S14	Detecting Irregular Patterns in IoT Streaming Data for Fall Detection	Mahfuz et al. 2018	----------	
S15	Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks	Marcu et al. 2016	----------	("Apache Flink" AND "Pattern")
S16	When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink	Verbitskiy et al. 2016	----------	

Table 8 Primary Studies Reporting NFRs in Big Data Systems

ID	**TITLE**	**AUTHOR(S)/YEAR**	**NFR**	**SEARCH STRING**
S1	Characterizing Big Data Software Architectures: A Systematic Mapping Study	Sena et al. 2017	Scalability, Performance, Modularity, Consistency, Security, Real-time operation, Interoperability, availability	("Reference Architecture" AND "Data Streaming System") ("Big Data" AND ("Software Architecture" OR" Reference Architecture" OR "Reference Model"))
S2	Detecting, Tracing, and Monitoring Architectural Tactics in Code	Mirakhorli et al. 2016	Security, Reliability, Performance	
S2	A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability	Carrez et al. 2017	Interoperability	
S3	Towards a Security Reference Architecture for Big Data	Moreno et al. 2018	Security	
S4	Simplifying Big Data Analytics Systems with a Reference Architecture	Sang, et al. 2017	Performance	
S5	Research on Reliability Evaluation of Big Data System	Cao et al. 2018	Reliability	
S6	Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review	Ullah et al. 2018	Performance, Accuracy, Scalability, Reliability, Security, Usability	

Broker Pattern:
This pattern is concerned with the structuring of DISTRIBUTED software SYSTEMs with decoupled components that interact by remote service invocations.

Context:
Your environment is a DISTRIBUTED and possibly heterogeneous SYSTEM with independent cooperating components.
Problem:
Sending requests to services in DISTRIBUTED SYSTEMs is hard. One source of complexity arises when porting services written in different languages onto different operating SYSTEM platforms. If services are tightly coupled to a particular context, it is time-consuming and costly to port them to another distribution environment or reuse them in other DISTRIBUTED applications. Another source of complexity arises from the effort required to determine where and how to deploy service implementations in a DISTRIBUTED SYSTEM. Ideally, services should interact by calling methods on one another in a common, location-independent manner, regardless of whether the services are local or remote.
Building a complex software SYSTEM as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability, and changeability. By partitioning functionality into independent components, the SYSTEM becomes potentially distributable and scalable.
Solution:
Use a federation of BROKERs to separate and encapsulate the details of the communication infrastructure in a DISTRIBUTED SYSTEM from its application functionality. Define a component-based programming model so that CLIENTs can invoke methods on remote services as if they were local.
SERVERs register themselves with the BROKER and make their services available to CLIENTs through method interfaces. CLIENTs access the functionality of SERVERs by sending requests via the BROKER. A BROKER's tasks include locating the appropriate SERVER, forwarding the request to the SERVER and transmitting results and exceptions back to the CLIENT. By using the BROKER pattern, an application can access DISTRIBUTED services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. In addition, the BROKER architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects. The BROKER pattern reduces the complexity involved in developing DISTRIBUTED applications because it makes distribution transparent to the developer. It achieves this goal by introducing an object model in which DISTRIBUTED services are encapsulated within objects. BROKER SYSTEMs, therefore, offer a path to the integration of two core technologies: distribution and object technology. They also extend object models from single applications to DISTRIBUTED applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.
Consequences:
The BROKER architectural pattern has some important **benefits:**
Location Transparency. As the BROKER is responsible for locating a SERVER by using a unique identifier, CLIENTs do not need to know where SERVERs are located. Similarly, SERVERs do not care about the location of calling CLIENTs, as they receive all requests from the local BROKER component.
Changeability and extensibility of components. If SERVERs change, but their interfaces remain the same, it has no functional impact on CLIENTs. Modifying the internal implementation of the BROKER, but not the APIs it provides, has no effect on CLIENTs and SERVERs other than performance changes. Changes in the communication mechanisms used for the interaction between SERVERs and the BROKER, between CLIENTs and the BROKER, and between BROKERs may require you to recompile CLIENTs, SERVERs or BROKERs. However, you will not need to change their source code. Using proxies and bridges is an important reason for the ease with which changes can be implemented.
Portability of a BROKER SYSTEM. The BROKER SYSTEM hides operating SYSTEM and network SYSTEM details from CLIENTs and SERVERs by using indirection layers such as APIs, proxies and bridges. When porting is required, it is therefore sufficient in most cases to port the BROKER component and its APIs to a new platform and to recompile CLIENTs and SERVERs. Structuring the BROKER component into layers is recommended, for example, according to the Layers architectural pattern. If the lower-most layers hide SYSTEM-specific details from the rest of the BROKER, you only need to port these lower-most layers, instead of completely porting the BROKER component.
Interoperability between different BROKER SYSTEMs. Different BROKER SYSTEMs may interoperate if they understand a common protocol for the exchange of messages. This protocol is implemented and handled by bridges, which are responsible for translating the BROKER-specific protocol into the common protocol, and vice versa.

Reusability. When building new CLIENT applications, you can often base the functionality of your application on existing services. Suppose you are going to develop a new business application. If components that offer services such as text editing, visualization, printing, database access or spreadsheets are already available, you do not need to implement these services yourself. It may instead be sufficient to integrate these services into your applications.

The BROKER architectural pattern imposes some **liabilities:**
Restricted efficiency. Applications using a BROKER implementation are usually slower than applications whose component distribution is static and known. SYSTEMs that depend directly on a concrete mechanism for inter-process communication also give better performance than a BROKER architecture, because BROKER introduces indirection layers to enable it to be portable, flexible and changeable.
Lower fault tolerance. Compared with a non-DISTRIBUTED software SYSTEM, a BROKER SYSTEM may offer lower fault tolerance. Suppose that a SERVER or a BROKER fails during program execution. All the applications that depend on the SERVER or BROKER are unable to continue successfully. You can increase reliability through replication of components.

The following aspect gives **benefits** as well as **liabilities**:
Testing and Debugging. A CLIENT application developed from tested services is more robust and easier itself to test. However, debugging and testing a BROKER SYSTEM is a tedious job because of the many components involved. For example, the cooperation between a CLIENT and a SERVER can fail for two possible reasons--either the SERVER has entered an error state, or there is a problem somewhere on the communication path between CLIENT and SERVER.

REFERENCES

James Carey and Brent Carlson. 2002. Framework Process Patterns: Lessons Learned Developing Application Frameworks. Addison-Wesley.

Ademar Aguiar, and Gabriel David. 2011. Patterns for effectively documenting frameworks. *Transactions on Pattern Languages of Programming II*, 79-124, Springer.

Len Bass, Paul Clements, and Rick Kazman. 2012. Software Architecture in Practice. *Addison-Wesley*.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. Pattern- Oriented Software Architecture: A System of Patterns. *John Wiley & Sons*, Volume 1.

Nick Rozanski, and Eoin Woods. 2012. Software Systems Architecture. Second Edition, *Addison-Wesley*.

Humberto Cervantes, Perla V. Elizondo, and Rick Kazman. 2013. A principled way to use frameworks in architecture design. *IEEE Software*, March/April, 46-53.

GemmaGrau, and Xavier Franch. 2007. A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures. *European Conference on Software Architecture (ECSA)*, pp 139-155.

Gemma Grau, and Xavier Franch. 2007. On the Adequacy of i* Models for Represeinging and Analysing Software Architectures. *To appear in Proceedings of the First International Workshop on Requirements, Intentions, and Goals in Conceptual Modelling, RIGiM.*

ISO/IEC 19505. Information technology – Object Management Group Unified Modeling Language (OMG UML), Infrastructure, (ISO/IEC 19505-2:2012). *Information technology – Object Management Group Unified Modeling Language (OMG UML)*, Superstructure (ISO/IEC 19505-2:2012).

Jeff Tyree and Art Akerman. 2005. Architecture Decisions: Demystifying Architecture. *IEEE Software*, pp. 19-27.

OMG: Systems Modeling Language (OMG SysML™), Version 1.3. Object Modelling Group 2012. (available on-line at http://www.omg.org/spec/SysML/1.3/).

Kent Beck and Ralph Johnson. 1994. Patterns Generate Architecture. *ECOOP '94 Proceedings of the 8th European Conference on Object-Oriented Programming*, pp 139-149, London, UK.

Jungwoo Ryoo, Rick Kazman, and Priya Anand. 2015. Architectural Analysis for Security. *IEEE Security & Privacy*, Vol 13, Issue 6.

Mehdi Mirakhorli. 2014. Preserving the Quality of Architectural Tactics in Source Code.

Mehdi Mirakhorli and Jane Cleland-Huang. 2016. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Transactions on Software Engineering*, Volume: 42, Issue 3, pp 205-220.

Mehdi Mirakhorli, A. Fakhry, A. Grecho, M. Wieloch, and Jane Cleland-Huang. 2014. Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp 739-742, Hong Kong, China.

Bruno Sena, L. Garces, A. P. Allian and Elisa Yumi Nakagawa. 2018. Investigating the Applicability of Architectural Patterns in Big data Systems. *Pattern Languages of Programs (PLoP)*, Portland, Oregon, USA.

Bruno Sena, Ana Paula Allian, and Elisa Yumi Nakagawa. 2017. Characterizing Big Data Software Architectures: A Systematic Mapping Study. *In Proceedings of SBCARS*, Fortaleza, CE, Brazil.

Eric Siu and Kwong Yu. 1995. Modelling Strategic Relationships for Process Reengineering. PhD. Thesis, University of Toronto.

Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. 2007. Pattern-Oriented Software Architecture. *Wiley*, the First Edition, Volume 4.

Han- Yuen Ong, Michael Weiss, and Ivan Araujo. 2003. Rewriting a Pattern Language to Make it More Expressive. 2003.

Gunter Mussbacher, Michael Weiss, and Daniel Amyot. 2007. Formalizing Architectural Patterns with the Goal-oriented Requirement Language. *Proceedings of the Fifth Nordic Conference on Pattern Languages of Programs*.

Matthias Meusel, K. Czarnecki, W. Kpf, and Daimler-benz Ag. 1997. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. *European Conference on Object-Oriented Programming ECOOP*, pp 469-510.

Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. *In Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

Wissem Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and Engelbert Nguifo. 2018. A Comparative Study on Streaming Frameworks for Big Data. *Latin America Data Science Workshop (LADaS) in conjuction with the 44th International Conference on Very Large Data Bases (VLDB)*.

Christos Vlassopoulos, Ioannis Kontopoulos, Michail Apostolou, Alexander Artikis, and Dimitrios Vogiatzis. 2016. Dynamic graph management for streaming social media analytics. *In Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 382–385. ACM.

Maribel Y. Santos, Jorge O. Esa, Carlos Costa, Joao Galvao, Carina Andrade, Bruno Martinho, Francisca V. Lima, and Eduarda Costa. 2017. A Big Data Analytics Architecture for Industry 4.0. *World Conference on Information Systems and Technologies*, pp 175-184, Volume 2.

Julio Moreno, Manuel A. Serrano, Eduardo F. Medina, Eduardo B. Fernandez. 2018. Toward a Security Reference Architecture for Big Data. DOLAP.

Francois Carrez, Tarek Elsaleh, David Gomez, Luis Sanchez, Jorge Lanza, and Paul Grace. 2017. A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability. *European Conference on Networks and Communications (EuCNC)*, Finland.

Ailing He, Jie Shen, Yaoliag Wang, and Li Liu. 2018. Research on the Fusion Model Reference Architecture of Sensed Information of Human Body for Medical and Healthcare IoT. *17th International Symposium on Distributed Computing and Applications for Business Engineering and Science*.

Go Muan Sang, Lai Xu, and Paul de Vrieze. 2017. Simplifying Big Data Analytics Systems with a Reference Architecture. Working Conference on Virtual Enterprises: Collaboration in a Data-Rich World, pp 242-249.

BlueTalon. 2016. Data-Centric Security Platform: Bringing Order to Data Security Chaos. http://bluetalon.com/data-centric_security/.

SQRRL.2014. Big Data and Data Centric Security. http://sqrrl.com/ media/Data-Centric-Security-WP-final-.pdf.

Eser Kandogan, Danny Soroker, Steven L. Rohall, Peter Bak, Frank V. Ham, Jie Lu, Harold Ship, and Chun-Fu Wang. 2013. A reference web architecture and patterns for real-time visual analytics on large streaming data, *Proceedings of SPIE - The International Society for Optical Engineering*.

Choong H. Kim, Kyungseo Park, J. Fu, and R. Elmasri. 2005. Architectures for Streaming Data Processing in Sensor Networks. *The 3rd ACS/IEEE International Conference onComputer Systems and Applications*, Cairo, Egypt.

Jasser Dhaouadi and Mehmet Aktas. 2018. On the Data Streaming Processing Frameworks: A Case Study. *3rd International Conference on Computer Science and Engineering (UBMK)*.

Go M. Sang, Lai Xu, and Paul de Vrieze. 2016. A Reference Architecture for Big Data Systems. *10th International Conference on Software, Knowledge, Information Management & Applications (SKIMA)*.

Dilpreet Singh and Chandan K. Reddy. 2015. "A survey on platforms for big data analytics. *Journal of Big Data*.

Sazia Mahfuz, Haruna Isah, Farhana Zulkernine, and Peter Nicholls. 2018. Detecting Irregular Patterns in IoT Streaming Data for Fall Detection. *IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*.

Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández. 2016. Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks. Cluster 2016 - The IEEE 2016 International Conference on Cluster Computing, Taipei, Taiwan.

IIya Verbitskiy, Lauritz Thamsen, and Odej Kao. 2016. When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink. *Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications*, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld).

Rui Cao, and Jing Gao. 2018. Research on Reliability Evaluation of Big Data System. *IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*.

Faheem Ullah, and Muhammad A. Babar. 2018. Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review. *ArXiv*.

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983.

Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and E. Romanova. 2007. Best practices for automated traceability Computer. vol. 40, no. 6, pp. 27–35.

Andrzej Zalewski. 2013. Modeling and Evaluation of Software Architecture. *Warsaw University of Technology Publishing Office*.