Evaluating Software Architecture Based on Their Implemented Patterns and Tactics

Hind Ahmad Ismail Bani Milhem

Thesis submitted to the
Faculty of Engineering
in partial fulfillment of the requirements for the degree of

Ph.D. of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

uOttawa

University of Ottawa
Ottawa, Ontario, Canada
Winter 2020

Abstract

Context: Software architecture plays a critical role in achieving system quality attributes. Therefore, evaluating a system's architecture with regard to desired quality requirements is very important. Architecture evaluation is an approach for assessing whether a software architecture can support the system needs, especially its quality attributes. Software architecture evaluation methods have been developed based on various characteristics and criteria such as the previous experience and domain knowledge of architects or developers, mathematical methods, features and scenarios, and testing. However, these methods may not be sufficient to reliably analyze certain quality attributes (i.e. performance, availability, and reliability). These methods also put little consideration on the architectural patterns and tactics used in the implementation, and the importance values of the desired quality attributes.

Objectives: This thesis proposes an architecture evaluation approach that considers satisfaction values of the quality attributes (Non-Functional Requirements) by the implemented patterns and tactics. The main objectives of this thesis are to provide:

- A way to connect a software implementation to quality attributes to support a software architecture evaluation based on its implemented architectural patterns and tactics. The evaluation considers the importance values of the quality attributes.
- Software architectures model in terms of their implemented architectural patterns and tactics taking into consideration the overlaps between the architectural patterns and tactics, and the importance values of the quality attributes. Such a model would provide a rationale about the satisfaction levels of given quality attributes and their trade-offs.

Method: In this thesis, I extract the implemented architectural patterns and tactics from a software architecture's source code and document them to connect the software architecture to quality requirements. I use a tool called Archie to extract the implemented

architectural patterns/tactics from software. I then document and model the patterns/tactics implemented by a software architecture and their impact on quality attributes using the Goal-oriented Requirements Language (GRL). Furthermore, I evaluate the GRL model of a software architecture by applying GRL/jUCMNav evaluation strategies to get the satisfaction values of the quality attributes. I validate the applicability and feasibility of our approach by applying it to different case studies from different contexts (big data systems, the healthcare system of systems, and build-automation systems). I compare the inferred quality attributes such as reliability, availability, performance, etc. to benchmark comparison results from the literature, and existing evaluation approaches.

Results: The satisfaction levels of the quality requirements by a set of architectural patterns and tactics of a software architecture, integrated with other criteria such as the importance values of the quality requirements, provide architects with a tool for evaluating different software architectures and documenting their rationale for assessing a software architecture. The three case studies show that our approach can be used to evaluate multiple software architectures and therefore, to identify strengths and weaknesses in different alternatives (i.e. alternative architectures, frameworks) and choose among them during the early design stages (i.e. cyber fusion center case study). Furthermore, it can be used to analyze, understand, and evaluate an existing implementation before future maintenance (i.e. HSH-SoS architecture case study). Additionally, our approach can be used to compare several implementations, based on specific quality attributes (i.e. Gradle and Maven case study). Finally, the modeling artifact should also enable faster evaluation with less efforts compared to the manual inspection of the source code and documentation of a software architecture.

Acknowledgment

First and foremost, I thank ALLAH, the almighty, for giving me the strength, knowledge, ability, and patience to work through all these years so that today I can stand proudly with my head held high.

I would like to gratefully thank my supervisor, Professor Stéphane Somé, for his assistance, the thoughtful comments and recommendations, guidance, and supervision throughout this research. His guidance helped me in all the time of research and writing of this thesis. I am gratefully indebted to his valuable comments and recommendations on this thesis.

I would also like to express the deepest gratitude and appreciation to Prof. Michael Weiss, of the department of Systems and Computer Engineering at Carleton University, whose valuable guidance, advice, and comments helped me very much throughout this research. He provided me with valuable support and gave generously his time, wisdom and experience. His support and help highly contributed to the success of this dissertation.

Many thanks to Prof. Daniel Amyot for his dedicated support and guidance, motivation, enthusiasm, and immense knowledge. His office was always open whenever I had a question about my research. His valuable comments and recommendations on this thesis helped me during the completion of this work.

I would also like to thank my Ph.D. committee members, Prof. Daniel Amyot, Prof. Hussein Al Osman, Prof. Neil Harrison, and Prof. Chung-Horng Lung For their brilliant and insightful comments, suggestions, and feedback, which improved my research from different angles and perspectives.

My heartfelt thankfulness goes to my lovely and always caring mother, Khadejah, and father, Ahmad, for their love, prayers, caring and sacrifices for educating and preparing me for my future. They have always encouraged me towards success and always directed me to take the right way. Without their love, encouragement and prayers, this work would never exist. I also express my thanks to my parents in law, Fatima, and Ali for their valuable encouragement, support, and prayers. Your prayer for me was what sustained me thus far.

Especially thank to my big sister Fayqa, whose unconditional love and care during my birth helped me focus on doing this work. She travelled from Jordan to Canada to take care of me and my kids. I also express my thanks to my sisters, brothers, sisters in law and brothers in laws for their support and valuable prayers.

Furthermore, my thanks go to all my friends and the people who have supported me to complete the research work directly or indirectly.

My acknowledgement would be incomplete without thanking the biggest source of my strength, my family, especially my husband, my very special person, Eng. Osama Milhem. I owe thanks to you for your continued and unfailing love, support and understanding during my pursuit of Ph.D. degree that made the completion of thesis possible. You were always around at times I thought that it is impossible to continue, you helped me to keep things in perspective. Trying to thank you with all possible languages in the earth is still not enough! I greatly value his contribution and deeply appreciate his belief in me. I appreciate my four daughters, Yara, Maya, Ellen, and Zain, for abiding my ignorance and the patience they showed during my thesis writing. Words would never say how grateful I am to you. I consider myself the luckiest in the world to have such a lovely and caring family, standing beside me with their love and unconditional support.

Dedication

This thesis is dedicated to:

- *My husband; Osama who has encouraged me all the way and whose encouragement has made sure that I give it all it takes to finish that which I have started. To my children Yara, Maya, Ellen, and Zain, without whom my life would be meaningless.*

- *To the souls of my father, my mother, and my father in law who raise and nurture me and fought for me to get the Ph.D. degree and be a brilliant professor.*

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
ABAS	Attribute-Based Architectural Styles
ADD	Attribute-Driven Design
ADL	Architectural Description Languages
AHP	Analytical Hierarchy Process
ALMA	Architecture Level Modifiability Method
ASG	Abstract Semantics Graphs
ATAM	Architecture Trade-off Analysis Method
CBAM	Cost Benefit Analysis Method
DSL	Domain-Specific Language
EBAE	Empirically Based Architecture Evaluation
FAAM	Family Architecture Analysis Method
FR	Functional Requirement
GRL	Goal-oriented Requirement Language
HDFS	Hadoop Distributed File System
EHR	Electronic Health Records
HIS	Health Information Systems
HSH-SoS	Health supportive Home –System of Systems
i*	I Star
JVM	Java Virtual Machine
KPIs	Key Performance Indicators
LQNs	Layered Queuing Networks
MARPLE-DPD	Metrics and Architecture Reconstruction Plugin for Eclipse Design Pattern Detection
NFR	Non-Functional Requirement
NFR-Framework	Non-Functional Requirement-Framework
PBAC	Role-Based Access Control
POM	Project Object Model
QA	Quality Attribute
RBAC	Policy-Based Access Control
SAAM	Software Architecture Analysis Method
SOA	Service-Oriented Architecture
SoS	System of Systems
SPE	Software Performance Engineering
SQL	Structured Query Language
SysML	System Modeling Language
UML	Unified Modeling Language
URN	User Requirements Notation
XML	Extensible Markup Language

Chapter 1. Introduction

This thesis presents a software architecture evaluation method based on the satisfaction levels of given Non-Functional Requirements (NFRs) (also referred to as quality attributes[1]) by the implemented architecture patterns and tactics of a software implementation. While architectural Patterns are solutions that apply to specific problems and their contexts, architectural tactics are design decisions that address given NFRs. The architecture patterns and tactics are extracted from source code. They connect the descriptive architectures of software implementations to quality requirements upon which software architecture evaluation can be made. The justification of the satisfaction levels of given NFRs by a software architecture can help architects and other stakeholders assess a software architecture.

1.1. Problem Statement

Software architectures have played a critical role in ensuring the quality attributes of a software system (i.e. security, performance, availability, usability, and maintainability). Furthermore, software architectures are created based on these quality attributes and impact them.

Software analysis and evaluation become a well-established practice inside the architecting community of the software systems. The development effort, time and costs of complex systems are considerably high. In order to assess the quality attributes of a software, the architects and the developers need methods and tools to support them during the evaluation process. Therefore, evaluating a software architecture and documenting the underlying rationale for the architecture become important tasks for system architects and developers. Many proposed architectures are usually available for a given system.

1 In this thesis, I will use both terms NFRs and quality attributes interchangeably

Architects need to evaluate and assess the potential of a proposed architecture that fulfill given quality attributes.

Various previous work [1][2][3][4][5][6][7] have addressed software architecture evaluation based on various characteristics and criteria such as the previous experience and domain knowledge of architects or developers, mathematical methods, features of the architectures, the deployability and the interoperability of the architectures, and how they perform (testing).

In this thesis, I focus on helping architects to assess software architectures. The proposed approach is based on a determination of the satisfaction levels of quality attributes by the patterns and tactics implemented by a software architecture. I address several shortcomings in current approaches:

- Lack of studies that connect a software architecture to qualities within a context of evaluating the software architecture.
- Lack of studies that evaluate software architectures based on the satisfaction levels of given Non-Functional Requirements by the implemented patterns and tactics of that architecture, considering the importance values of the NFRs.
- Lack of studies which extract the implemented patterns and tactics from the source code of a software architecture to be used to connect the architecture to qualities
- Lack of studies that model software architectures in terms of their implemented patterns and tactics.

The above-mentioned challenges motivate the need to find an informed way to connect a software architecture to qualities, which need to be satisfied, to be able to evaluate the software architecture based on its implemented patterns and tactics. These patterns and tactics are extracted from the software implementation. This solution should provide the ability to represent software architectures, their patterns and tactics, their quality attributes, design decisions, the participated actors, and the general goal of a system in a comprehensive model. Such a model would provide a rationale about the candidate software architectures in terms of the satisfaction levels of the NFRs impacted by their implemented patterns and tactics. The rationale should provide the ability to evaluate these candidate software architectures and therefore, to identify strengths and weaknesses in different alternatives (i.e. software architectures, frameworks) and help choosing among

them during the early design stages. Furthermore, the model can be used to compare several tools, based on specific quality attributes. Additionally, it can be used to analyze, understand, and evaluate an existing implementation of a software or an existing system before future maintenance or improvement of the system. The solution should also be able to help identify potential risks in the form of quality attributes trade-offs, and enable faster evaluation with less efforts than traditional manual approaches.

1.2. Motivation

Several software architecture evaluation approaches have been proposed [1][2][3][4][5][6][7]. However, none of the existing approaches provides a way to evaluate automatically software architectures' intrinsic quality attributes. Quality attributes such as performance, security, modifiability, reliability, and usability are significantly impacted by the architecture of a software system [8]. Stakeholders such as architects and developers need to understand the design of software architecture in terms of quality attributes. For example, they need to understand if they will achieve deadlines in real-time systems, what kind of modifications or changes are supported by the design and how the system will respond in case of a failure.

Architectural patterns and tactics are reusable building blocks for software development. They are characterized in terms of factors that affect (both positively and negatively) various quality attributes [8]. Knowing the characteristics, strengths, and weaknesses of each architecture pattern and tactic is necessary in order to determine whether they promote or impede the satisfaction of quality requirements in a system.

Giving the following two hypotheses:

- Software architectures are created with an objective of achieving quality attributes, based on patterns and tactics that impact these quality attributes.
- A software implementation inherits the quality aspects of the patterns and tactics used as part of its implementation.

I have been motivated to use architectural patterns and tactics to connect software implementations to the satisfaction of quality requirements. Software architecture

evaluation based on that connection can be made. This motivation led us to the research questions in the next section.

1.3. Research Questions

Based on our hypothesis that a software architecture inherits from quality attributes associated to the patterns and tactics used in its implementation, I formulate the following primary research question:

Primary Research Question:
"How can I connect a software implementation to quality attributes in an objective (concrete) way to be able to evaluate the software architecture?"

Given that, a software architecture is created based on patterns and tactics, and it inherits from the qualities' aspects of those patterns and tactics used in its implementation, I could formulate the following secondary questions from the mentioned primary question:

Secondary Research Questions:

1. **How can I determine (extract) the patterns and tactics used by a software architecture?**
2. **How can I evaluate software architectures based on their implemented patterns/tactics?**

1.4. Research Method

In this thesis, an iterative approach of research and validation has been followed. The validation was performed to evaluate the direction of research and validate our approach. This approach is adapted from the "Design science in information systems" research method, proposed by Hevner [9][10], and it relies on iterative improvements to the research. To be more specific, I followed the steps presented in Figure 1, adapted from [10].

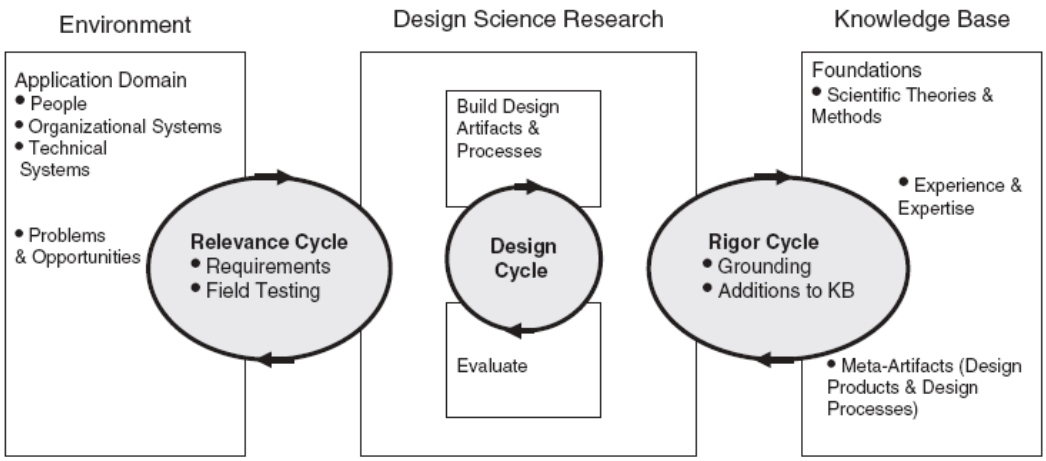


Figure 1 Design Science Research cycles (Adapted from [10])

In the following, I discuss the three research cycles which are adapted to the needs of this research:

Step 1: Relevance Cycle

- The idea for this research started with our involvement in an industrial project supervised by Prof. Michael Weiss at Carleton University. The objective of this project was to make small companies more resilient to cyber threats by building a cyber fusion center. The cyber fusion center aims to aggregate information about security events and architecture-related information from the installed application stacks, identify attack patterns and architectural flaws, generate and deploy new sensor rules, and identify areas of the application source code that need to be patched. The project work was distributed between several academic (Carleton University and Concordia University) and industry partners (AvanTech and eGloo). In this project, architects needed to evaluate the architecture of several candidate open source development frameworks to be able to choose among these candidate frameworks to build the cyber fusion center. The selected framework was to 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 (in order to detect attacks and attack patterns). The selected framework

should satisfy the non-functional requirements of the cyber fusion center. Given the lack of documentation, an objective evaluation approach was needed to be able to determine which of the candidate frameworks was better suited. A high degree of automation of the approach was required for it to be effective given the size and complexity of the frameworks.

- Background review: In this step, I conducted some initial background review to make ourselves more familiar with the area of research, pinpoint the problem areas and opportunities in our area of interest.
- Problem Definition: Once a better understanding of the area of research was gained in the background review step; I defined the topic and general scope of the thesis.
- Research design: In this step, I defined the steps of the Rigor and Design cycles, including the tools to be used in our approach, the examples to be used for defining and validating our approach, and validation assessments to be performed at the same time.

Step 2: Rigor Cycle

Literature review and validation: I started by searching for literature reviews on the topic of interest. I also found existing related research (approaches) and analyzed them. There was no approach that completely solves the problem. Most of the existing approaches referred to the importance of the software architecture evaluation during the processes of the development and reusing of applications or part of applications. Because architectural patterns and tactics are reusable building blocks for software development (including frameworks) and are characterized in terms of factors that affect (both positively and negatively) various quality attributes, I came up with our hypothesis to connect software implementation to quality requirements using patterns and tactics upon which a software architecture evaluation can be performed.

Step 3: Design Cycle

- Initial definition of the approach: In this step, I defined the approach by identifying its main steps, inputs, and outputs (deliverables). I also identified the potential contributions.

- Evaluation using case studies. In this step, I applied the approach to case studies from different contexts (big data systems, the healthcare system of systems, and build-automation systems).
- Further validation: I further validated the approach by comparing the inferred quality attributes such as reliability, availability, performance, etc. with benchmark comparison results from the literature. Furthermore, I validated the approach by comparing it to other existing selection approaches.
- The above validation steps were also used as feedback loops for previous cycles.

1.5. Thesis Contributions

This section gives the highlights of the thesis chapters, together with their *major* and *minor* contributions towards solving the problems discussed in Section 1.1. The value and impact of these contributions, together with future work, will be further discussed in the conclusion.

The major contributions are:

1- Software architecture evaluation approach that helps assess an implementation based on the uncovering, analysis and modeling of the architectural patterns and tactics used in the code. This contribution is described in Chapters 4, 5, and 6.

2- An inventory of reusable models for tactics and patterns, gaining in the creation of a model for the software architectures. This contribution is discussed in Chapter 5.

The minor contributions are:

1- Adapting Mirakhorli's approach [11][12][13] by adding additional tactics and patterns to the Archie tool. This contribution has been made by performing a literature review to find the most common patterns and tactics in a software implementation in a specific context. Then, I added these patterns and tactics to the Archie tool [11][12][13] to be detected for a software architecture. I also validated the results of the Archie tool by manually hunting the occurrences of the patterns

and tactics detected by Archie, in the source code of the software architecture. This contribution is discussed in Chapter 4.

2- Review of existing software architecture evaluation approaches. This contribution is discussed in Chapter 2.

3- Review of existing techniques for modelling a software architecture. This contribution is discussed in Chapter 2.

4- Review of existing techniques for connecting software implementation to quality attributes. This contribution is discussed in Chapter 2.

5- Review of existing techniques for determining patterns and tactics in an implementation. This contribution is discussed in Chapter 2.

6- Literature review to find the most relevant patterns, tactics, and NFRs for systems in a specific context. This contribution is discussed in Chapter 4.

1.6. Publications

The following is the list of the accepted and submitted publications resulting from the research done for this thesis:

Accepted:

1. **Milhem, H.**, Weiss, M., and Somé, S.: "Extraction of Architectural Patterns from Frameworks and Modeling their Contributions to Qualities". In: 26th *Proc of Conf. on Pattern Languages of Programs (PLoP 2019).* Ottawa, October 2019.

   This paper presents a *method* for extracting architectural patterns from frameworks and modelling their contributions to qualities with a case study from a cyber security center project (discussed in Chapters 4 and 5).

2. **Milhem, H.**, Weiss, M., and Somé, S.: "Modeling and Selecting Frameworks in terms of Patterns, Tactics, and System Qualities". In: *32th Proc of*

International Conference on Software Engineering and Knowledge Engineering (SEKE 2020). USA, July 2020. (Best Paper Award).

This paper gives an overview of the approach proposed in this thesis. It proposes a *methodology* to compare the quality attributes of frameworks. The methodology relies on two major steps: a) determining patterns and tactics implemented in the frameworks, b) modeling and evaluating the frameworks in terms of their implemented patterns and tactics. (discussed in Chapters 4, 5, 6, and 7).

Submitted:

3. **Milhem, H.**, Weiss, M., and Somé, S.: "Modeling and Selecting Frameworks in terms of Patterns, Tactics, and System Qualities". *International Journal of Software Engineering and Knowledge Engineering (ijSEKE 2020).* 2020.

   This paper gives an overview of the approach proposed in this thesis. It proposes an approach to compare the quality attributes of frameworks. The approach relies on three major steps: a) determining patterns and tactics implemented in the frameworks, b) modeling the frameworks in terms of their implemented patterns and tactics, c) evaluating the frameworks based on their implemented patterns and tactics (discussed in Chapters 4, 5, 6, and 7).

4. **Milhem, H.**, Weiss, M., and Somé, S.: "A Methodology for Frameworks-Selection based on the impact of their implemented Architectural Patterns and Tactics on Quality Attributes". *Journal of Information and Software Technology*.

   This paper gives an overview of the approach proposed in this thesis. It proposes a *methodology* to compare the quality attributes of frameworks. The methodology relies on two major steps: a) determining patterns and tactics implemented in the frameworks, b) modeling and evaluating the frameworks in terms of their implemented patterns and tactics (discussed in Chapters 4, 5, 6, and 7).

5. **Milhem, H.**, Weiss, M., and Somé, S.: "Extraction of Architectural Patterns from Frameworks and Modeling their Contributions to Qualities". *Transaction of Pattern Language oriented Program (TPLoP) journal*, 2019.

   This paper presents a *method* for extracting architectural patterns from frameworks and modelling their contributions to quality attributes (discussed in Chapters 4 and 5).

1.7. Overview of the Approach

Our general approach **S**oftware **A**rchitecture **E**valuation based on **P**atterns and **T**actics (SAEPT) has three main steps, which are outlined as follow.

1- Determining patterns and tactics implemented in a software architecture. This step is discussed in Chapter 4.

In this step, I extract the implemented architectural patterns and tactics from software architectures' source codes. I adapted Archie to be used to determine patterns and tactics from source code. The adaption has been done by adding additional patterns and tactics to Archie to be detected in a new context/domain.

2- Modelling software architectures in terms of their implemented patterns and tactics. This step is discussed in Chapter 6.

In this step, I model the patterns and tactics used by a software architecture and their impact on quality attributes, in order, to build an inventory of the GRL models of patterns and tactics, this is discussed in Chapter 5. I use the inventory to model architectures in terms of their implemented patterns and tactics as discussed in Chapter 6.

3- Using the model to evaluate software architectures. This step is discussed in Chapter 6 together with the second step of the approach.

In this step, I evaluate the models and, therefore, evaluate software architectures based on the satisfaction levels of the given NFRs considering importance values

of these NFRs given by the architecture evaluators. This step includes the following sub-steps:

a) Evaluate the models of the software architectures.

   I evaluate the model of the software architectures by applying different evaluation strategies, to the model using the selected modelling language. This sub-step is discussed in Section 6.4.1.

b) Evaluate the software architectures.

   I evaluate the software architectures based on the satisfaction levels of the NFRs by patterns and tactics considering the importance values of these NFRs. This sub-step is discussed in Section 6.4.2.

In this thesis, I have a specific realization of the approach based on the use of specific tools. For step 1, I used the Archie tool [11] [12] [13]. Our discussion about Archie and other tools is presented in Sections 3.5 and 4.3. For step 2, I used the Goal-oriented Requirement Language (GRL) [14]. Our discussion about GRL and other modeling languages is presented in Sections 3.6 and 5.1.1. For step 3, I used GRL/jUCMNav [10]. Our discussion about the evaluation process using GRL/jUCMNav is presented in Chapter 6.

I illustrate the main steps, the key activities which have been followed to perform the main steps, the roles involved in case this approach meant to be used by others, and the deliverable of each step in Table 1.

Table 1 The general steps of the approach, key activities, and deliverables

Main Steps	**Key Activities**	**Roles**	**Deliverables**
1. Determining patterns and tactics implemented in a software architecture. This step is discussed in Chapter 4.	-Adapt the tool by adding tactics and patterns to be detected in a specific context/domain.	Approach configurator	Candidate set of patterns and tactics for a software architecture
	-Run the tool on the source code of a software architecture to search patterns and tactics.	Architect	

	-Validate the results by reviewing the literature (code/website/documentation) of a software architecture to look for references to specific patterns and tactics, and then hunting for the occurrences of those patterns and tactics in a source code	Architect	A table of the detected patterns and tactics from the literature and the source code of a software architecture
	-Determine the overlaps between patterns and tactics.	Architect	Tables of the overlaps' results
2. Modelling software architectures in terms of their implemented patterns and tactics. This step is discussed in Chapter 6.	- Extract consequences and liabilities from pattern descriptions and the associated rationale.	Approach configurator	List of the extracted NFRs, the contributions of patterns/tactics on those NFRs, and the associated rational
	- Calculate the contribution values of the patterns and tactics to given NFRs.	Approach configurator	List of the contribution values of patterns and tactics to given NFRs
	-Derive GRL models from the descriptions of the patterns and tactics.	Approach configurator	Models inventory
	-Identify the most architecturally significant NFRs to be considered in the design of frameworks in the determined context. This has been done by performing a literature review to find the most architecturally significant NFRs.	Approach configurator	List of systems in the determined context referred to NFRs with Table of NFRs relevant to those systems
	-Calculate the importance values of the most architecturally significant NFRs.	Architect	List of the important values of the determined NFRs

	-Model a software architecture in terms of all its detected patterns and tactics to satisfy relative NFRs.	Architect	Software architecture model
3. Using the Model to evaluate software architectures. This step is discussed in Chapter 6.	-Apply evaluation strategy to the model using the selected modelling language.	Architect	Evaluated models of software architectures
	-Evaluate the software architectures based on the satisfaction levels of given NFRs considering the importance values of the NFRs of a software architecture.	Architect	A rationale about each software architecture in terms of its patterns and tactics so an architect can assess the architecture

1.8. Chapter Summary

In this chapter, I introduced the context of the problem addressed in this thesis. I assert that this thesis is an effort to provide an approach for the evaluation of software architectures in terms of their patterns and tactics. I also discuss other findings such as NFRs, architectural patterns, architectural tactics, as elements supporting the main hypothesis. The discussion of these findings is presented in Chapter 3.

The rest of this document is organized as follows. In Chapter 2, I present other related work. In Chapter 3, I introduce basic concepts related to this thesis. In Chapter 4, I introduce a method of determining the patterns and tactics which are implemented in software architecture. In Chapter 5, I explain the modeling method of the software architectures in terms of their patterns and tactics. In Chapter 6, I present the evaluation approach of software architectures based on their patterns and tactics considering the importance values of NFRs. In Chapter 7, I discuss the experimental validation of the approach using real case studies. The results, the validation, and the evaluation of our approach are presented in Chapter 8. In Chapter 9, I present the conclusion and future work of this thesis.

Chapter 2. Literature Review

In this chapter, I discuss work related to our research from the academic literature. In Section 2.1, I introduce our review methodology. Section 2.2 discusses approaches to connect software implementation to quality attributes. Section 2.3 discusses approaches to determine patterns and tactics in an implementation. Section 2.4 reports on techniques for modelling software architecture in terms of their implemented patterns and tactics. Section 2.5 discusses approaches for evaluating software architecture. Finally, Section 2.6 provides a summary of the Chapter.

2.1. Review Methodology

In this thesis, I investigate the work proposed in the literature for modeling software architectures in terms of their implemented patterns and tactics in support of software architecture evaluation. In particular, I formulate our research queries according to the following questions reflecting our research questions presented in Chapter 1:

- Question 1: “Are there existing approaches/work aiming to connect software implementation to quality attributes?”
- Question 2: “Are there existing approaches/work aiming to determine (extract) the patterns and tactics implemented by a software?”
- Question 3: “Are there existing techniques aiming to model software architectures in terms of their implemented patterns and tactics?”
- Question 4: “Are there existing approaches/work aiming to evaluate software architectures based on their implemented patterns/tactics”

The general query used to answer Question 1 is:

```
("Connect*" OR "Link*" OR "Attach*")
AND ("Quality Attribute" OR "QA"
        OR "Non Functional Requirement" OR "NFR")
AND ("Software" OR "Implementation" OR "System"
        OR "Framework" OR "Design")
```

AND ("Software Architecture")

The general query used to answer Question 2 is:

("Determin*" OR "Identif*" OR "Extract*" OR "Detect*")
AND "Pattern" AND "Tactic"
AND ("Implementation" OR "Architecture" OR "Source Code"
OR "Framework" OR "System" OR "Design")
AND "Software Architecture"

The general query used to answer Question 3 is:

("Model" OR "Formaliz*")
AND ("Software" OR "System" OR "Framework" OR "Design")
AND ("Pattern" OR "Tactic") AND "Software Architecture"

The general query used to answer Question 4 is:

("Evaluat*" OR "Assess*" OR "Analy*")
AND ("Software" OR "Implementation" OR "Framework"
OR "System" OR "Design")
AND "Software Architecture"

The search strings were executed in the data libraries Scopus and Google Scholar. Such libraries were selected because they index research works published in other important computer science academic libraries, such as ACM Digital Library, SpringerLink, and IEEE Xplore. The search was constrained to Title-Abstract-Keywords. Regarding Question 4, I used a wide query to cover all architecture evaluation approaches, then I used manual filtering to exclude irrelevant papers. For all questions, I also extended the initial selection with relevant articles cited in the selected papers (via snowballing).

Based on the above search method, I categorize the related works that were retrieved into the following research areas:

1) Existing approaches aiming to connect software implementation to quality attributes (Question 1). These approaches are discussed in Section 2.2.
2) Existing approaches aiming to determine patterns and tactics in an implementation (Question 2). These approaches are discussed in Section 2.3.
3) Existing techniques for modeling software architectures in terms of their implemented patterns and tactics (Question 3). These techniques are discussed in Section 2.4.
4) Existing approaches for evaluating software architectures (Question 4). These approaches are discussed in Section 2.5.

2.2. Existing Approaches Aiming to Connect Software Implementation to Quality Attributes

I only found one approach that aiming to connect software implementation to quality attributes. An example is Tvedt et al. [15] which proposes a process for inspecting the architecture of an implemented system using metrics. The objective of the approach is to ensure that properties such as maintainability are preserved after a system has been updated. The proposed process compares the actual architecture (after an update) to the planned architecture and identifies potential deviations. Tvedt et al. observe that the implementation of a system may involve architectural styles and design patterns. Detecting the occurrences of these styles and patterns helps to identify the actual architecture of an implemented system. However, the paper does not elaborate on how to detect these architectural styles and patterns or how these elements connect to quality attributes.

2.3. Existing Approaches Aiming to Determine Patterns and Tactics in an Implementation

The literature includes a large number of works on the determination of patterns or tactics in an implementation. An analysis of these works shows that this is a challenging problem. Although in our approach I only consider architectural patterns, in Section 2.3.1, I discuss approaches aimed at determining patterns in general (architectural and design patterns)

from an implementation. Section 2.3.2 presents approaches aimed to determine tactics in an implementation.

2.3.1. Approaches to Determine Patterns in an Implementation

Much work has been done in the area of pattern detection/determination, yet some key issues still exist. One of these issues is that there is still no consensus on the information that shows the existence of patterns in source code. Development of consistent definitions of patterns is, in fact, the fundamental demand for the pattern research community. Indeed, there are different interpretations and implementation variants of patterns. Another issue is that pattern detection approaches are usually language dependent and support in the best case only a small number of programming languages due to the variety of features for implementing patterns across languages and so requires a successful approach to handle language dependent variants of patterns. Furthermore, patterns vary in complexity. Some patterns are easily identified due to their unique structure, while it is more challenging for others.

In this section, I present several patterns detection/recognition approaches. Note that, the references Cervantes et al. [1], Johnson [16], Meusel et al. [17], and Beck and Johnson [18] discuss detection methods for architectural patterns, while the references from Rasool and Mäder [19] to Guéhéneuc et al. [47] present detection methods for design patterns.

Architectural patterns, that I provided definitions for in Chapter 3, can be detected using several approaches such as matching methods between the provided services of an application and its patterns/tactics [1][16]. Cervantes et al. in [1] extract patterns from a framework implementation by applying a mapping process between the patterns in the framework and those patterns 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). In comparison to our work, their approach does not consider the patterns and tactics instantiated as a selection criterion from the beginning of the process. Their work also does not provide any details on how to represent implementations in terms of their patterns and tactics and how to compare and select the best fit framework for a given set of quality attributes. Their extraction of the tactics is

manual using a mapping process between the patterns in a framework and those patterns which are employed in architecture design.

Architectural patterns can also be determined using pattern instantiation. Meusel et al. in [17] describe the process of pattern 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.

Matching methods between a problem statement and the applied patterns in the architecture is used to detect architectural patterns by Beck and Johnson in [18]. They extract patterns from the problem statement of architecture to document frameworks. The method is applied to HotDraw. The authors describe the problem statements of the HotDraw framework and then recognize the patterns used to solve these problems.

Rasool and Mäder [19] propose an open and extensible detection approach for design patterns [20]. This approach is based on pattern definitions composed of reusable feature types. These pattern definitions are elementary properties likely to reoccur across different pattern types (e.g., an aggregation relation between two classes or a method return and the instance of the class that contains it) [21]. The patterns are then detected by identifying these feature types using multiple search techniques such as Structured Query Language (SQL) [22], regular expressions, and source code parsers. Mirakhorli et al. [21] used a similar approach to detect design patterns in the source code of several open-source projects.

Shi and Olsson [23] use lightweight static program analysis techniques to capture program intent in order to recognize patterns from Java source code. They reclassified all Gang of Four (GoF) design patterns [20] in the context of reverse engineering. Their tool recovers all GoF patterns from source code with the exception of Prototype, Iterator, and Builder.

Tsantalis et al. in [24] propose a design pattern detection methodology based on the degrees of similarity between pattern structure graphs and programs structure graphs. They developed a Java tool that automates their methodology and generates a list of the detected pattern instances. The program uses a Java bytecode manipulation framework that provides detailed information about the static structure of the system. Matrices representing the system under study are constructed according to that information.

The approach presented in [25] proposes a matrix-based approach. The authors used matrices and weights to recover design patterns. Their approach uses prime numbers to encode the characteristics of systems and patterns (i.e. structure, behavior, and semantics of design patterns) into matrix and weights. Classes in a system are represented as rows and columns, and relationships between a pair of classes are represented using a value of a corresponding cell in the matrix. The recovery of design patterns is performed by matching matrices and weights using arithmetic computations. The authors performed experiments on different case studies to recover design patterns.

Yu et al. [26] propose an approach to detect design patterns based on graph isomorphism. The authors represent a system design in one graph and design patterns in other graphs. To detect instances of design patterns, all the candidate classes that correspond to the pattern classes are identified in the system graph. These candidate classes are then connected into graphs and checked for isomorphism with sub-graphs in the system graph. Those isomorphic sub-graphs are regarded to be corresponding to instances of the design patterns.

Dong et al. [27] adopt a template matching method to detect design patterns. Information about both patterns and systems is encoded into matrices. They calculate the normalized cross correlation between pattern and system matrices. A normalized cross correlation shows the degree of similarity between a design pattern and part of a system. A Design Pattern Miner (DP Miner) tool [25] developed by the authors implements their method. The proposed method in Gupta et al. [28] is another design pattern detection method based on calculating the normalized cross correlation between pattern and system matrices.

Zanoni et al. [29] apply machine learning techniques for design pattern detection. They developed a Metrics and Architecture Reconstruction Plugin for Eclipse Design Pattern Detection (MARPLE)-DPD, based on a combination of graph matching and machine learning techniques. The architecture of (MARPLE)-DPD consists of two primary modules, a Joiner which extracts (possibly) all the pattern instances in a software system, and a Classifier which classifies as correct or incorrect the instances detected by the Joiner. The Joiner uses rule-matching. The rules are very general and consider fundamental traits

of pattern structures. Other design pattern detection methods based on machine learning techniques are presented in Dwivedi et al. [30] and Uchiyama et al. [31].

Arcelli and Cristina [32] present an approach to automate design pattern detection using classification and data mining techniques based on sub-components (set of structures summarized from source code). They use MARPLE [29] to help in design pattern recovery.

Niere et al. [33] propose an approach to recognize instances of design patterns semi-automatically in source code. They use Fujaba [33]. Fujaba seeks a combination of inter-class relationships for pattern detection. Using a bottom-up search, when partial inter-class relationships matching a pattern are found, Fujaba switches to a top-down search strategy to try to confirm the presence of the pattern. This strategy speeds up the search and reduces the false positive rate.

Various approaches such as DP++ [34], SPOOL [35], and Osprey [36] extract interclass relationships (class inheritance, interface hierarchies, modifiers of classes and methods, types and accessibility of attributes, method delegations, parameters and return types) from C++ source and store to a database. Patterns are then recovered by querying the database. SOUL [37] is a logic inference system used to recognize patterns based on inter-class-based code idioms and naming conventions.

Guéhéneuc et al. [38] use program metrics such as size, cohesion, and coupling, as well as a machine learning algorithm to fingerprint roles of a pattern's participating classes. These fingerprints are learnable facts for a pattern constraint solver. The fingerprints are quantitative signatures of design motifs roles that can be used to reduce efficiently the number of classes playing potentially a role in a design motif [38].

In [39], design pattern detection is accomplished via the integration of two existing tools – Columbus [40] and Maisa [41]. The method combines the extraction capabilities of the Columbus reverse engineering system with the pattern mining ability of Maisa. First, C++ code is analyzed by Columbus. Then the facts collected are exported to a clause-based design notation understandable to Maisa. The instances are then searched for matches to design pattern descriptions.

Ferenc et al. [42] combine the Columbus reverse-engineering framework with the Maisa architectural metrics analyzer [41] to build a pattern recognizer. Maisa analyzes software at the design level and reports results including on anti-patterns [43]. Balanyi and

Ferenc [44] use the Columbus schema for extracted abstract semantics graphs (ASG) [44] and recover patterns based on graph comparison. Antoniol et al. [45] extract inter-class relationships and then uses software metrics to reduce search space. Ferenc et al. [46] use machine learning methods to refine pattern mining by marking pattern instance candidates returned by the matching algorithm either as true or false instances. Machine learning techniques are used to train a pattern recognition tool. Each pattern is defined with a set of predictors whose values are used in the learning process.

Guéhéneuc and Antoniol [47] present a multilayered semiautomatic approach for design pattern detection from Java source code. The approach uses static analysis to detect relationships. Dynamic analysis based on trace analysis techniques is used to compute exclusivity and lifetime relationships for aggregation and composition relationships.

2.3.2. Approaches Aiming to Determine Tactics in an Implementation

Unlike approaches for patterns detection, there are fewer approaches on tactics detection. An example is Mirakhorli and Cleland-Huang [11][12][13]. This approach relies primarily on information retrieval and machine learning techniques for discovering tactics in code. The approach consists on training a classifier to recognize specific terms that commonly occur across implemented tactics. The probabilities of these terms (the probability that a particular term identifies a class associated with a tactic) are determined using specific mathematical equations. The approach is implemented in a tool called Archie, which I use in this thesis. I present more details about Archie in Chapter 3. Mirakhorli and Cleland-Huang demonstrated their approach by applying it to Hadoop Distributed File System (HDFS). In this thesis, I adapted Archie to help us to determine tactics and patterns in source code. For more details about our adaption process, I refer to Chapter 4.

Zhu et al. [48] also present an approach to extract tactics from design patterns in order to improve the software architecture evaluation process. Their approach is based on observing that there is a relationship between patterns and tactics so that tactics are used in patterns. They determine these related tactics from the solution and quality consequence sections of pattern documentation. The approach is illustrated with an example where two tactics are extracted from the Data Transfer Object Factory (DTOFactory) pattern description [49].

Ryoo et al. [50] discuss three approaches for extracting tactics: deriving new tactics from the existing ones, decomposing an existing architectural pattern into its constituent tactics, and extracting tactics that have been misidentified as patterns. The authors only focus on extracting security tactics. Their work aims to create updated tactics hierarchies that refine and improve the existing ones. They also presented several examples of tactics.

2.4. Techniques for Modelling Software Architectures in Terms of their Implemented Patterns and Tactics

In this section, I present existing work from the literature on modelling software architectures in terms of their implemented patterns and tactics.

Grau and Franch [2] propose an approach for modelling architectures using the i* framework [2][51]. They use these models to generate guidelines to create alternative architectures and define metrics to evaluate the resulting alternative models. They applied their approach to a Home Service Robot case study.

Zalewski [3] presents a survey about the state of the art in modelling and evaluating software architectures. The author describes three main approaches: models of software structure, architectural decisions, and models of the architecture description. The author enumerates a number of semi-formal models used to model software architecture such as block diagrams models [52], the Unified Modeling Language (UML) [53], the System Modeling Language (SysML) [54], and Archimate [55]. Various models for documenting architectural decisions are also discussed and compared, such as textual models [56], and a comprehensive model by Zimmerman et al. [57]. The Early Architecture Evaluation Methods developed for the evaluation of large-scale system architectures at the inception stage of development is also covered.

The Goal-oriented Requirement Language (GRL) is considered by Mussbacher et al. [14] as a tool to achieve system goals, stakeholder concerns, alternative means of achieving goals, and the rationale for goals and alternatives. They used GRL to formalize patterns forces and their interactions in a way to guide the selection of the most suitable patterns among many alternatives.

The NFR-framework is another technique used to model architectures. For example, Araujo and Weiss in [58] applied the NFR-framework as a technique to represent

relationships between design decisions and non-functional requirements explicitly. They found the NFR-framework as being a modelling technique that allows the expression of rich and structured relationships between NFRs. This permits the expression of pattern languages with precise semantics.

Mohsin et al. [59] present the results of a literature review about the architecture modelling approaches for System of Systems (SoS). They provide a taxonomy for modelling SoS obtained as a result of the literature review. They used Architecture Description Languages (ADL) to deal with and model the SoS dynamic architectures.

Table 2 summarizes the modelling techniques used to model software architectures.

Table 2 Existing modelling techniques

Authors and Ref	**Modelling technique used OR discussed**	**Architecture element(s) modeled**
Grau and Franch [2]	i* (i star)	Architectures
Zalewski [3]	Block diagrams models, Unified Modeling Language (UML), System Modeling Language (SysML), and Archimate	Software architectures
Mussbacher et al. [14]	Goal-oriented Requirements Language (GRL)	Patterns forces and their interactions
Weiss and Araujo [58]	NFR-Framework	Design decisions and non-functional requirements
Mohsin et al. [59]	Architectural Description Languages (ADL)	Systems-of-Systems Dynamic Architectures

2.5. Approaches for Evaluating Software Architecture

Several approaches on evaluating a software architecture have been developed to assess the potential of a proposed/chosen architecture based on different characteristics and criteria. However, none of the reviewed approaches provides a way to evaluate a software architecture's quality attributes automatically. Software architecture evaluation approaches can be divided into four main categories: experience-based, simulation-based, mathematical modeling, and scenario-based. In section 2.5.1, I present the experience-based approaches for evaluating software architectures. Section 2.5.2 discusses the simulation-based evaluation approaches. In section 2.5.3, I report on the mathematical modeling evaluation approaches. The scenario-based evaluation approaches are discussed in section 2.5.4. Section 2.5.5 presents some other related work specifically focusing on the documentation of architectures, which can be used to analyze and evaluate architectures. I also present a comparison between these approaches and our proposed approach in Table 3.

2.5.1. Experience-Based Evaluation Approaches

The experience-based evaluation approaches are based on the previous experience and domain knowledge of developers or architects [60]. For example, the Empirically Based Architecture Evaluation (EBAE) by Lindvall et al. [61] is an experience-based evaluation method for evaluating the maintainability of a re-designed software system developed more or less in-house compared to its previous version. The authors define and use a number of architectural metrics to evaluate and compare the architectures. The basic steps in the EBAE process are 1) select a perspective for the evaluation, 2) define/select metrics, 3) collect metrics, and 4) evaluate/compare the architectures [61].

The Attribute-Based Architectural Styles (ABAS) [62] is another experience-based evaluation method used to evaluate various quality attributes such as performance or maintainability. It enables the analysis of different quality aspects of software architectures. ABAS is a general process that can be used to analyze several quality attributes, given that quality models are provided for the relevant quality attributes [62].

According to Cervantes et al. in [1], architectures can be evaluated using patterns. These patterns can be determined by applying a mapping process between the patterns in

an implementation and those patterns which are employed in architecture design. They also mention that the mapping between the provided services of a framework and the patterns can be used to determine patterns in a framework. These patterns can be used for framework selection. The selection can be based on architecture drivers (such as the team's level of knowledge of a framework, or the framework's maturity).

2.5.2. Simulation-Based Evaluation Approaches

Simulation-based evaluation approaches rely on a high-level implementation of some or all of the components in the software architecture [6]. For example, Layered Queuing Networks (LQNs) [7] are used for the simulation-based evaluation of software *performance* [7]. The application of LQNs relies on the transformation of the architecture into a layered queuing network model. The model describes the interactions between components in the architecture and the processing times required for each interaction. The creation of the models requires detailed knowledge of the interaction of the components, together with behavioral information (i.e. execution times or resource requirements) [7].

Argus-I [63] is a simulation-based evaluation method that evaluates a number of aspects of an architecture design. This approach allows to perform structural analysis, static behavioral analysis, and dynamic behavioral analysis, of components [63]. It also can perform dependence analysis, interface mismatch, model checking, and simulation of architecture [63]. Argus-I uses a formal description of a software architecture and its components together with StateCharts that describe the behavior of each component. The described architecture can then be evaluated in terms of performance, dependence, and correctness [63].

SAM [64] is a formal systematic methodology for software architecture specification and analysis. The methodology is used to evaluate the performance of a system. It defines software architectures and their properties, and then perform formal analysis of them using formal methods. SAM uses time Petri nets and temporal logic to support an executable software architecture specification [64]. Hierarchical architectural decomposition is used to facilitate scalable software architecture specification.

2.5.3. Mathematical Modeling Evaluation Approaches

These approaches use mathematical proofs and methods for evaluating quality attributes. They share similarities with simulation-based methods and can be combined with to more accurately estimate the performance of components in a system. Layered Queuing Networks (LQNs) [7] discussed in the previous section, can also be considered for mathematical modeling evaluation methods.

Software Performance Engineering (SPE) [65] uses a mathematical modeling evaluation method for building performance into software system. It relies on two different models of the software system, a software execution model and a system execution model. The software execution model models the software components, their interaction, and the execution flow. The resource requirements, such as execution time, memory requirements, and I/O operations, are also included in the software execution model. The software execution model predicts the performance without taking the contention of hardware resources into account [65].

2.5.4. Scenario-Based Evaluation Approaches

Scenario-based methods evaluate a particular quality attribute by creating a scenario profile that forces a very concrete description of the quality requirement [4]. The Software Architecture Analysis Method (SAAM) [4], the Architecture Trade-off Analysis Method (ATAM) [4], the Cost Benefit Analysis Method (CBAM) [4][66], the Architecture Level Modifiability Method (ALMA) [67], and the Family Architecture Analysis Method (FAAM) [68] are the most common scenario-based evaluation methods. In the following, I provide details for two of these methods: SAAM and FAAM. The interested reader can find the comparisons between these scenario-based methods in Table 3.

The Software Architecture Analysis Method (SAAM) is the first widely scenario-based evaluation method. It was created [4] to assess the *modifiability* of an architecture. Zhu [69] used the scenario-based evaluation method in his thesis to evaluate framework-based systems for the selection of reasoning frameworks.

The SAAM method consists of six steps [4]:

1) *Develop scenarios*: This step includes a brainstorming exercise with the scope of identifying the type of activities that the system must support. These activities

together with modifications that the stakeholders can anticipate are grouped in so called system scenarios [4].

2) *Describe architecture(s):* In this step, the candidate architectures are presented and understood by the participants. The architectures must also indicate the static representation of the system (components, their interconnections and the relationships) as well as the dynamic behavior of the system [4].
3) *Classify and prioritize scenarios*: The scenarios are classified into direct and indirect scenarios. A direct scenario is supported by the candidate architecture because it is based on requirements from which the system has been evolved [4]. An indirect scenario is a sequence of events whose realization involves minor or major changes in the architecture.
4) *Individually evaluate indirect scenarios:* The scenarios are evaluated by architects to demonstrates how the scenario would be realized by the architecture in case of a direct scenario. In case of an indirect scenario, the architect describes how the architecture would need to be changed to accommodate the scenario [4].
5) *Assess scenarios interaction:* To avoid the interaction of different scenarios, in case two or more scenarios request changes over the same component(s), the affected components need to be modified or divided into sub-components [4].
6) *Create overall evaluation: In this final step,* a weight is assigned to each scenario in terms of its relative importance to the success of the system. Based on these weights, architectures can be compared and alternatives for the most suitable architecture can be proposed [4].

The Family-Architecture Assessment Method (FAAM) focuses on two related quality attributes: *interoperability and extensibility*. It includes six steps [68]:

1) *Define the assessment goal:* This step is needed to determine the goal of the assessment.
2) *Prepare system quality requirements:* In this step, the system's requirements are identified and prioritized by stakeholders.

3) *Prepare architecture:* the architecture representation is prepared for presentation and assessment against the stakeholders' requirements. Guidelines are also provided to architects for representing the architectural views.
4) *Review/refine artifacts:* all the requirements and architectural views are reviewed and refined. The goal of this step is to come to an agreement on these requirements and architectural views. The step also includes clarifying the business or logical constraints that may affect the assessment continuation.
5) *Assess architecture conformance:* The architecture description is verified against the specified requirements with focus on the ability and easiness to integrate or to satisfy change-cases specified in Step 2 [68].
6) *Report results and proposals:* the assessment results are recorded and communicated back to the stakeholders, in this step. Based on the results of Step 5 the facilitator together with the architecting team draws the lessons learned from the assessment exercise [68].

2.5.5. Other Related Work

In this section, I present other works related to ours. The following approaches specifically focus on the documentation of architectures. They can be used to analyze and evaluate architectures.

Johnson [16] defines frameworks as a set of patterns and other components. He also described the relationship between frameworks and patterns. In his view, frameworks represent and contain several patterns, since they are implemented several times. The paper gives examples of frameworks and their representation using patterns. It introduces the concept of using patterns to document frameworks. Our work builds on that by using the documentation of software architectures in terms of patterns and tactics to evaluate these software architectures.

Aguiar and David [70] describe the idea of documenting object-oriented frameworks in terms of a set of patterns that capture proven solutions to recurrent problems. They describe the considered patterns in terms of their name, context, problem, and solution. These patterns are intended as a starting basis for those willing to learn how to document a framework. Compared to our work, their work introduced the idea of frameworks

documentation using patterns but did not discuss the evaluation process of their architecture.

Beck and Johnson [18] document the HotDraw framework. They describe the problem statements of the HotDraw framework and then recognize the patterns used to solve these problem statements. Our work builds on that by using the documentation of implementations in terms of patterns for evaluation.

Van Heesch, Avgeriou, and Hilliard [71] introduce a documentation framework for architecture decisions. The framework consists of four viewpoints, a Decision Detail viewpoint, a Decision Relationship viewpoint, a Decision Chronology viewpoint, and a Decision Stakeholder Involvement viewpoint. The framework is used for addressing reocurring or typical concerns related to architecture decisions. The authors show that decision views facilitate communication between stakeholders, support technical architecture reviews, and enable the reuse of architecture decisions.

Sena et al. [72] conduct a Systematic Mapping Study to gather the main characteristics, basic requirements, quality attributes, design patterns, modules, and organization of big data architectures. The authors identify eight requirements that could directly impact big data architectures and five modules for big data architectures. I use the results of this work to determine the main quality requirements in our case studies as discussed in Chapters 4 and 7.

Sena et al. [73] 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 determine 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. I use these results to determine the main quality requirements and the determined patterns in Section 4.2.

Ryoo et al. [74] describe architectures in terms of tactics. These tactics are collected by performing interviews with architects. They focus only on tactics. However, interviews can also be used to extract patterns from an architecture.

Meusel et al. [17] describe the process of pattern instantiation by assigning roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a concrete

design. The goal of this method is to extract patterns from a design. Table 3 summarizes the differences between the existing approaches and our approach based on various criteria.

Table 3 Comparisons between existing approaches and our approach

Approach	Category	Consider Patterns	Consider Tactics	Extraction method of patterns/tactics	The degree of automation of the identification of patterns and tactics	Consider the modelling of the architecture
SAAM [4]	Scenario-based	NO	NO	----------	----------	NO
ATAM [4]	Scenario-based	NO	NO	----------	----------	NO
CBAM [4][66]	Scenario-based	NO	NO	----------	----------	NO
ALMA [67]	Scenario-based	NO	NO	----------	----------	NO
FAAM [68]	Scenario-based	NO	NO	----------	----------	NO
LQN [7]	Simulation-based, Mathematical modeling	NO	NO	----------	----------	Yes
ARGUS-I [63]	Simulation-based	NO	NO	----------	----------	NO
SAM [64]	Simulation-based	NO	NO	----------	----------	NO
EBAE [61]	Experience-based	NO	NO	----------	----------	NO
ABAS [62]	Experience-based	NO	NO	----------	----------	NO
SPE [65]	Mathematical modeling, Simulation-based,	NO	NO	----------	----------	Yes
Cervantes et al. [1]	Documentation-based	YES	NO	Attribute-Driven Design (ADD)	Non-automated	NO

Johnson [16]	----------	Yes	NO	----------	----------	NO
Mirakhorli and Huang [11] [12] [13]	----------	NO	Yes	Archie tool	Semi-Automated	NO
Aguiar and David [70]	----------	Yes	NO	----------	----------	NO
Beck and Johnson [18]	Documentation-based	Yes	NO	problem statement of an architecture	Non-automated	NO
Heesch, Avgeriou, and Hilliard [71]	----------	Yes	NO	----------	----------	NO
Sena et al. [72]	NA	Yes	NO	Literature review	Non-automated	NO
Sena et al. [73]	NA	Yes	NO	Literature review	Non-automated	NO
Ryoo et al. [74]	Documentation-based	NO	Yes	Interviews	Non-automated	NO
Meusel et al [17]	Documentation-based	Yes	NO	Pattern instantiation	Non-automated	NO
Zhu [69]	Scenario-based	Yes	NO	Manually	Non-automated	NO
Our approach (SAEPT)	**Patterns-Tactics based, Modeling-based**	**Yes**	**Yes**	**Archie tool and manual search**	**Semi-automated**	**Yes**

2.6. Chapter Summary

In this chapter, I introduced work related to our research. First, I discussed approaches that aiming to connect software implementation to quality attributes. Second, I introduced approaches aiming to determine patterns and tactics in an implementation. Third, I present existing techniques for modeling software in terms of their implemented patterns and tactics. Fourth, I discussed approaches aiming to evaluate software architectures.

Chapter 3. Background

In this thesis, I deal with several concepts coming from different areas of software engineering. I introduce the background terminology and the existing work related to key concepts relevant to our work, including the concepts of the architectural development process, development frameworks, architecture drivers, and architectural elements. I also introduce the tools and the methods which have been used in this thesis, such as Archie, the Goal-oriented Requirement Language (GRL), and Ong's method. These concepts and tools are described in general as well as according to the specific considerations taken by this thesis.

Architecture drivers include quality attributes (NFRs), functional requirements (a function of a system or components of a system, something that the system should do) [16], context, and constraints (restrictions on any provided solution) [16]. In this thesis, I only consider NFRs (quality attributes) from the architecture drivers. Architecture elements include architectural patterns and architectural tactics. While architectural patterns express high-level design decisions, an architectural tactic is a design strategy that addresses a quality attribute [75]. The use of patterns encourages consistency and increases the speed of development. A knowledge of the patterns used in an implementation (i.e. a framework) also helps understand and leverage it [76].

3.1. Architectural Development Process

According to [77][78], "architecture development can be defined as both a process and a discipline that aids the development of mission-effective systems". The architecture development process is a cyclic and iterative process that ensures concordance between views in the architectural description while ensuring that all essential data relationships are captured to support a wide variety of analysis tasks. The Attribute-Driven Design (ADD) [77] is one of the design methods focuses specifically on achieving quality attributes through the selection of different types of structures and their representation through views.

There are seven steps to the ADD method, as shown in Figure 2 [77]. The application of our work for framework selection can be situated within Step 4 of the ADD process.

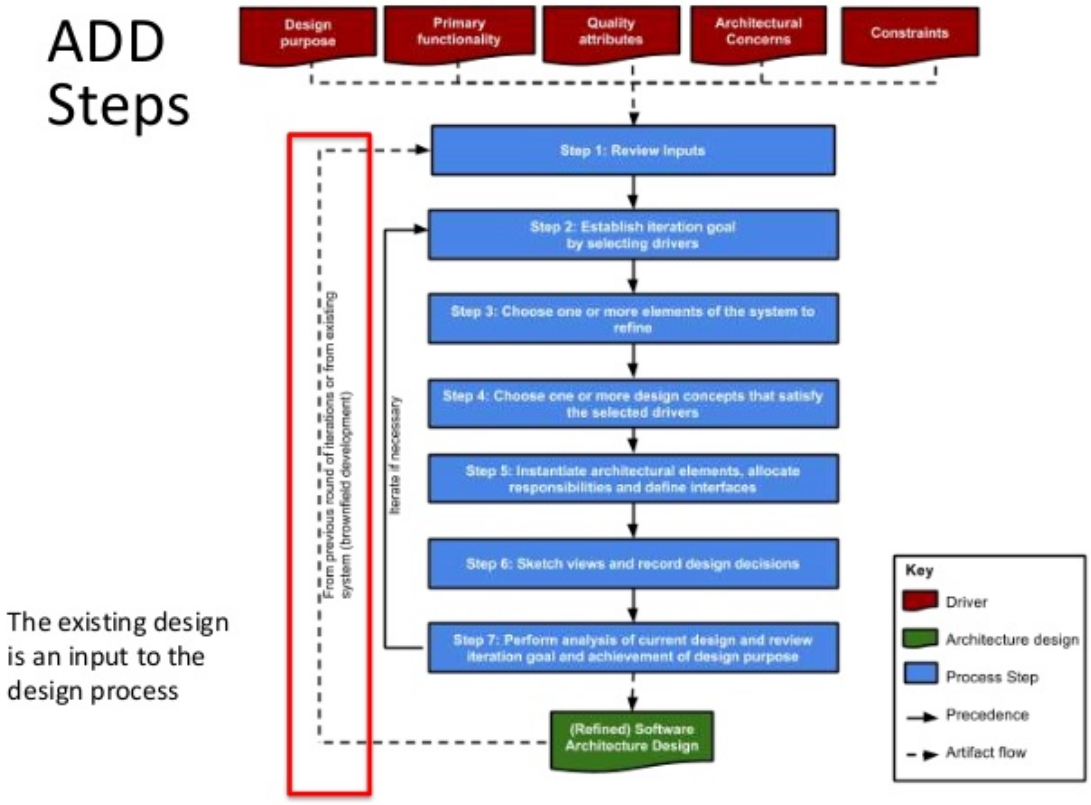


Figure 2 ADD Steps (Adapted from [77])

Following are details of the ADD steps [77].

***Step 1: Review inputs*:** In this step, the availability and the correctness of the inputs to the design process should be checked. First, architectures should be built with a specific purpose, whether the intent is business process reengineering, system acquisition, user training, or any other intention [77][78]. Second, before starting to describe an architecture, an organization must determine the issue(s) the architecture is intended to explore, the questions the architecture is expected to answer, and the interests of the users. Third, the types of analysis that are expected to be performed is another intent that should be considered as well. This will make the architecture development effort more efficient and the resulting architecture more balanced and useful.

***Step 2: Establish the iteration goal by selecting drivers*:** In this step, the design round (architecture activities performed within a development cycle) is performed in series of design iterations where each iteration achieves a particular goal. The perspective content of the architecture can be determined. The items to be considered include the scope of the architecture, which includes the activities, functions, and timeframes. The context of the architecture effort, the operational scenarios, the economic situation, and the capabilities of specific technologies are other items that also should be considered in this step.

***Step 3: Choose one or more elements of the system to refine*:** Architectural structures consist of a set of interrelated elements. These elements are generally obtained by refining other elements identified in previous iterations. Elements refinement can be performed by either decomposing elements into finer-grained elements or combining elements into coarse-grained elements. The elements chosen to be refined are involved in the satisfaction of specific architecture drivers (a set of requirements that influence an overall architecture, see Section 3.3).

Step 4: Choose one or more design concepts that satisfy the selected drivers: This step requires to analyze and choose among design concepts alternatives. Frameworks are a category of these design concepts. One of the potential applications of our work is to help architects analyze frameworks according to their degree of satisfaction of quality attributes and guide the selection.

Step 5: Instantiate architectural elements, allocate responsibilities, and define interfaces: After choosing one or more design concepts, elements must be instantiated out of the selected design concepts. For example, if the ***Pipes and Filters*** pattern is selected in Step 4, then decisions such as the format of data that each filter receives and sends, as well as the type of filters must be made in this step. After instantiation, responsibilities should be allocated to the elements. For example, each layer in the ***Layers*** patterns should be allocated to specific responsibilities (specific tasks). Then, the elements that have been instantiated also need to be connected, to allow them to collaborate with each other. This is done by defining the interfaces of components.

Step 6: Sketch views and record design decisions: In this step, all the findings resulting from the previous steps are documented. This includes sketching views (a preliminary

version of documentation for the created structures). This also includes recording design decisions and their rationale.

Step 7: Perform analysis of current design and review iteration goal and achievement of design purpose: This step includes analyzing the partial design created within the previous steps. This involves reviewing the sketches of the views and the recorded design decisions. It is suggested to have reviewers different from the designers with different perspectives. This can help find issues as well as bugs in both the code and architecture.

3.2. Development Frameworks

A development 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. It is used to solve or address complex issues, and usually consists of a set of tools, materials or components [78].

Development frameworks are related to patterns and tactics in that frameworks instantiate (implement) these concepts. The patterns and tactics that a framework instantiates allow quality attributes to be satisfied [1]. The Hibernate framework is a popular example of object-relational mapping framework [1]. Another example is Spring (an extensive framework for developing enterprise applications) [1]. Apache Storm [79], Apache Metron [80], Apache Flink [81], Apache Spark [82], and Angular [83] are all other examples of the development frameworks. Some of these frameworks, which are of interest to this thesis (i.e. Apache Storm, Apache Flink, and Apache Spark), are introduced in Chapter 7.

3.3. Architectural Drivers

Architectural drivers are considerations that need to be made for the software system that are architecturally significant. They include a set of requirements that influence the overall architecture [16][75]. They drive and guide the software architecture design. The designed software architecture needs to satisfy these architectural drivers. *Non-Functional Requirements (NFRs), Functional Requirements (FR), Context,* and *Constraints* are architecture drivers. Architecture elements discussed in Section 3.4, are selected in order to support architecture drivers.

3.3.1. Non-Functional Requirements (NFRs)

Non-functional requirements, Quality Attributes (QAs) or *quality requirements*, are characteristics that are required from the system. They are qualifications of the functional requirements or the overall product such as performance, security, usability, and reliability [16][75]. These qualifications should be considered with the functions of the system. No system's function can stand without due consideration of NFRs. For example, if a system has a function that the pressing of a green button should make an options dialogue to appear, then the performance NFR might describe how quickly that dialogue should appear.

Non-Functional Requirements describe the circumstances that surround a particular problem [75]. They also define the scope and the system's desired functionality and quality properties. The satisfaction of NFRs depends on a specific scale, the specific context of the scenario, and given system with a particular input and output.

3.3.2. Functional Requirements (FRs)

Functional Requirements (FRs) are functions of a system or components of a system, or something that the system should do [16]. Functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures. A functional requirement describes a particular behavior of a system when certain conditions are met, for example: “Send email when a new customer signs up”.

Functional requirements are supported by non-functional requirements. Generally, functional requirements are expressed in the form of “system must do something” while non-functional requirements take the form of “system shall be a quality”. The implementation of functional requirements is detailed in the system design, whereas non-functional requirements are addressed in the system architecture [16].

3.3.3. Context

A *context* describes the circumstances that surround a particular problem [75]. It defines the scope and the system's desired functionality and quality properties. Examples of statements of architecture contexts are: “constructing a system from a collection of services distributed across multiple servers” or “a system that requires transformation streams of discrete data items, from input to output”.

According to [75], *technical context, project life cycle context, business context,* and *professional context* are types of contexts.

The *technical context* [75] refers to the technical role played by the software architecture in a system or systems. Examples of technical contexts are web-based systems, object-oriented systems, service-oriented systems, cloud-based systems, and social-networking systems. In contrast, the *project life cycle context* [75] refers to how the software architecture development relates to the other phases of a software development life cycle. The *business context* [75] concerns how a software architecture affects the business environment of an organization. Finally, the *professional context* [75] expresses the role of a software architect in an organization or development project. This role is characterized by the skills, experience, knowledge and duties performed by architects.

3.3.4. Constraints

Constraints are restrictions on any provided solution [16]. They are design decisions with zero degree of freedom [1]. Examples include the requirement to use a certain programming language or to reuse an existing module.

In software architecture design, constraints come in two basic flavors - *technical* and *business* [1]. *Technical constraints* are fixed design decisions that absolutely cannot be changed. Examples are programming languages (ex. Java, .Net, and Ruby), operating systems or platforms (such as Windows and Unix/Linux).

Business constraints are fixed decisions to business nature, that impact architectural structures indirectly [1]. Examples of business constraints are schedules where the final delivery date is fixed for business reasons, and fixed budgets.

3.4. Architecture Elements

Architecture elements include *architectural patterns* and *architectural tactics*.

3.4.1. Architectural Patterns

Architectural Patterns are solutions that apply to specific problems and their contexts. Examples of architectural patterns include the ***Broker*** pattern [84][85], the ***Layers*** pattern [84][85], and the ***Pipes and Filters*** pattern [84][85]. *Architectural patterns* express high-level design decisions and describe high-level structures and behaviors [16][75]. They

provide a solution for a specific problem in a specific context to satisfy the functional requirements, non-functional requirements, and constraints of a system. An architectural pattern may have *positive* or *negative* impacts on NFRs. For example, in the ***Layers*** pattern, if an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts, resulting in a positive contribution to *Reusability*. In contrast, if high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers and maybe transformed several times. Therefore, the ***layers*** pattern could have a negative contribution to the *Efficiency* requirement.

3.4.2. Architectural Tactics

Architectural tactics are design decisions that affect the achievement of NFRs response They are used to address quality requirements [75]. Examples of tactics include *"Heartbeat"* [75], *"Ping/Echo"* [75], *"Authentication"* [75], and *"Authorization"* [75]. While, *architectural pattern* expresses high-level design decisions, an *architectural tactic* is a design strategy that addresses a particular NFR [16][75]. In general, a tactic implementation includes structure and behavior and can influence architectural patterns in several ways when implemented together. Tactics can be implemented in the same structure as architectural patterns or require changes to the structure and behavior of architectural patterns.

3.5. The Archie Tool

Archie is an Eclipse plugin. It is used to detect architectural tactics, monitors related Java code, and to notify developers when they modify architecturally significant parts of the code [11][12][13]. It is built to help automate the creation and maintenance of architecturally relevant trace links between code, architectural decisions, and related requirements. The links are then used to monitor significant areas of the code actively and to generate timely user notifications describing underlying architectural decisions [11][12][13]. Archie has the following capabilities [11][12][13]:

- A detection engine that identifies sections of the code implementing architectural decisions. The detection engine uses classifiers trained for code detection. Archie

also has an interactive viewer that allows a user to browse through code snippets returned by the detection engine.

- An annotated code viewer capable of highlighting architecturally significant parts of the code.
- Visualization features used to generate views of architectural tactics, their relationships to design rationales and requirements. These features are also used to create global views of architectural decisions.
- Features that allow a user to bypass the automated detection process, and manually mark-up sections of code as being architecturally significant.
- An event engine that constantly monitors changes to the code in the background, notifies the user when he/she starts to modify sensitive areas of the code and displays information about the underlying architectural decisions.

Archie is released as an open-source project in GitHub under the name Archie-Smart-IDE. Figure 3 shows Archie as a plugin tool in the Eclipse platform with the areas of interest of this thesis.

The area of determined patterns and tactics shows the detected patterns and tactics for a framework. These patterns and tactics are shown with snippets of code where the related terms of a pattern/tactic are detected. The code viewer area shows the whole code where the related terms of a pattern/tactic are detected. The detection of patterns and tactics can be configured in the automatic detection area according to a specific threshold. The candidate frameworks area shows all the candidate frameworks that need to be analyzed by Archie.

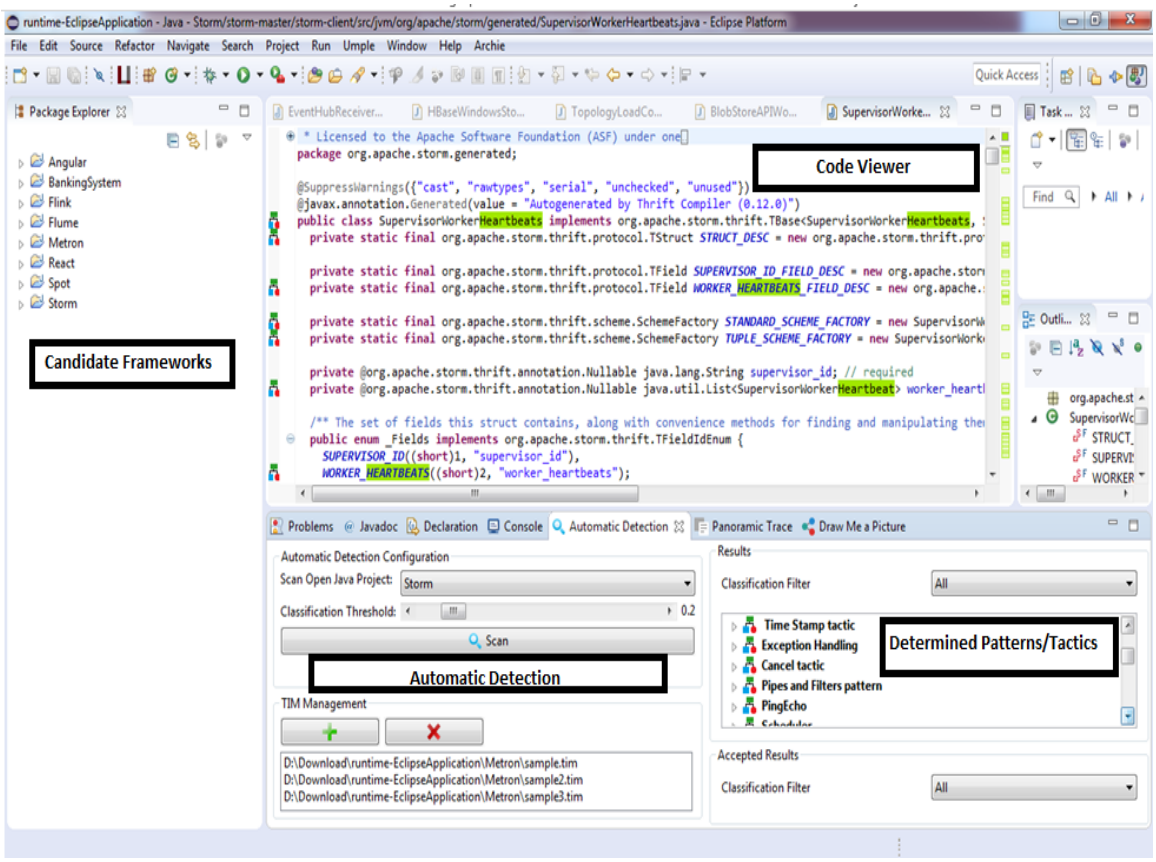


Figure 3 Archie tool: Eclipse plugin

Archie has an integrated tactic detector. The tactic detector is a customized version of a previously developed *NFR-classifier* [11][12][13]. Detection involves three phases, *preparation, training,* and *detection*. The *preparation phase* includes preprocessing the data by using standard information retrieval techniques. These include the removal of non-alpha-numeric characters, stemming words to their morphological roots, and removal of 'stop' words. The remaining terms are then transformed into a vector of terms [11][12][13]. In the *training phase*, Mirakhorli and Huang trained a classifier in *Archie* to recognize specific terms that occur commonly across implemented tactics and calculate the weights of the terms (the probability that a particular term identifies a class associated with a tactic). In the *detection phase*, the probability (score) that a given Java class is associated with a tactic is calculated. The calculation depends on the sum of all probabilities of the terms that are contained in a Java class divided by the sum of probabilities of all the trained-on terms.

Archie allows a user to run or re-run the tactic detector against source code to generate a list of candidate tactics of a framework.

In this thesis, I used *Archie* to extract the implemented patterns and tactics in the source code of a framework. Although a manual search of the implemented patterns and tactics in a framework is possible, it is not practical for large frameworks. Different alternative methods have been used in the literature, such as:

a. *Archie* [11][12][13] (a tool that relies primarily on information retrieval and machine learning techniques for discovering tactics in code).

b. Matching methods between the provided services of a framework and its patterns/tactics [16].

c. *Pattern instantiation* (assigning the roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a practical design) [70].

d. Matching method between the problem statement of architecture and the applied patterns [18].

e. *Matrix-based approach* [25].

f. *Graph isomorphism-based approach* [26]. The system design is represented in one graph and the design patterns, to be recovered, in another graph.

g. Various tools, such as *Design Pattern Miner (DP Miner)* [25], *Architecture Reconstruction Plugin for Eclipse (MARPLE)-DPD* [29], *Fujaba* [33], *DP++* [34], *SPOOL* [35], *Osprey* [36].

h. *Logic inference* (SOUL [37]) based on inter-class-based code idioms and naming conventions.

i. The integration of existing tools – *Columbus* [40] and *Maisa* [41].

j. *Multilayered semiautomatic approach* [47].

k. Determination of the tactics from the solution and quality consequence sections of a pattern's documentation [69].

I chose *Archie* among the previously mentioned methods because according to Mirakhorli's work [11], *Archie* is an extensible tool so that additional tactics can be added by defining terms related to these tactics with their weights. *Archie* needs to be extended, by adding additional patterns and tactics, to detect both patterns and tactics. Additionally, *Archie* is not just about finding that tactics are implemented, but it is also about finding where these tactics are implemented and monitoring changes to their implementation. It has an interactive interface so I can run more than one architecture, and it is a plugin of the Eclipse platform. *Archie* also has been tested on several systems ranging from 1,000 to 20,000 Java files. One of the case studies is based on Hadoop, a large system with three major subsystems and many hundreds of programs. *Archie* makes the extraction process faster by decreasing time and effort spent searching the patterns and tactics and their related terms in the documentation, websites, and source codes of architectures.

Archie has several limitations. One of these limitations is that *Archie* is limited to Java. This limitation restricts our case studies to be only Java-based systems. *Archie* also only considers thirteen tactics from three quality attributes in Java-based systems. These tactics are related to *Security, Reliability,* and *Performance*.

The *Security* tactics are:

- Policy-Based Access Control (PBAC)
- Role-Based Access Control (RBAC)
- Kerberos
- Audit-trail
- Session Management
- Authenticate.

The *Reliability* tactics are:

- Checkpoint
- Heartbeat
- Ping/Echo
- Active Redundancy
- Load Balancing.

The *Performance* tactics are:

- Resource Scheduling
- Resource Pooling.

Archie is an extensible tool so that additional tactics can be added by defining terms related to tactics, with their weights. However, Mirakhorli and Huang do not consider patterns in the *Archie* tool. This is due to the observation that patterns are typically described in terms of several aspects, such as intent, structure, behavior, and sample code. Unlike tactics, a pattern usually consists of multiple elements forming its structure, of relationships between these elements and of a described behavior of the elements [21].

I opted to use *Archie* for detecting architectural patterns and tactics, despite the acknowledgement of Archie's authors point of view concerning the inadequacy of the tool for patterns. I felt that *Archie* offered the best approach given our work constraints. Archie has been shown to work well for tactics [11][12][13]. I explored the possibility of combining with one of the above-mentioned tools for patterns. However, most of these tools are meant to detect design patterns rather than architectural patterns. Design patterns generally have a well-defined structure in term of classes and associations making approaches such as graph matching suitable. Other design patterns detection tools are based on machine learning techniques that necessitate a substantial number of code samples for training. In order to use any of those tools, adaptation would be required for architectural patterns. Depending on the tool, this would involve modelling of architectural patterns implementation, bottom-up or top-down inspection of architectural patterns code, machine learning training, etc.

Archie offers a simple extension mechanism by which the capability to detect additional tactics can be added by specifying related terms. I chose to exploit this feature to adapt *Archie* for the detection of both patterns and tactics. This provides a benefit of only using a single tool. I recognize however limitations to our approach from the point of view of the precision of the results. But since in addition to detecting tactics (and patterns), Archie identifies the area of the code where these elements are suspected, I am able to manually validate the results and eliminate False Positives as discussed in Chapter 4. A proper approach for architectural patterns detection will be needed for complete automation.

I add more tactics and patterns to *Archie* while avoiding the need for the training stage to determine the related terms and their weights as done by Mirakhorli and Huang. This is because the training stage requires several code samples which I do not have for practical reason. Alternatively, I recognize the terms from the description of the patterns and tactics. This is because I believe that there is a connection between the terminology used to describe tactics and patterns in their documentation and the terminology in the corresponding source code. For instance, an implementation of the Broker Pattern is likely to include terms such as *broker, producer, consumer* present in the pattern description, as variables, methods, classes names or in comments. This is supported by numerous studies such as [86], [87], [88], and [89] that have shown that code written by developers, often uses mnemonics for identifiers. The analysis of these mnemonics can help to link high-level concepts with program concepts, and vice-versa [86][87].

Finally, during the first step of our approach, I am in fact, only interested in knowing whether tactics/patterns are implemented, but not where these patterns/tactics are implemented. I am only interested in where these patterns/tactics are implemented to manually validate the effective presence of detected tactics and patterns.

3.6. Goal-oriented Requirement Language (GRL)

GRL [14][90] is a goal-oriented requirement language used for modelling goal-oriented and intentional concepts (related to Non-Functional Requirements, quality attributes, and reasoning about alternatives). It consists of *intentional elements* (e.g. goals, softgoals, and resources), *intentional links* (e.g. decomposition of type AND, OR, or XOR, contribution of type qualitative or quantitative, and dependency links), and *actors* (e.g. stakeholders or systems themselves). I use GRL to model implementations in terms of the patterns and tactics that they use and their impact on NFRs. The elements of the GRL notation used are shown in Figure 4.

Different modelling languages have been used to model requirements. Examples include: The Goal-oriented Requirement Language (GRL) [14], the NFR-framework [58][91][92][93], i* (i-star) framework [2][51].

In this thesis, I chose the GRL because it enables us to evaluate and compare the impact of different design choices on quality attributes. GRL furthermore, is a part of an international standard (User Requirements Notation – URN) [14], enables the modelling

of stakeholders and their goals, supports Key Performance Indicators (KPIs) for quantitative reasoning, and supports evaluation strategies and propagation algorithms to evaluate the satisfaction of goals and actors under selected conditions [90]. Giving quantitative contributions of patterns and tactics helped us calculate the satisfaction of NFRs. Some of the GRL limitations are related to the use of decomposition links [94][95], the lack of modularity [96][97] and the cognitive fitness of its graphical syntax [97]. For more details about using GRL models and their benefits over other popular goal notations, see [14][90].

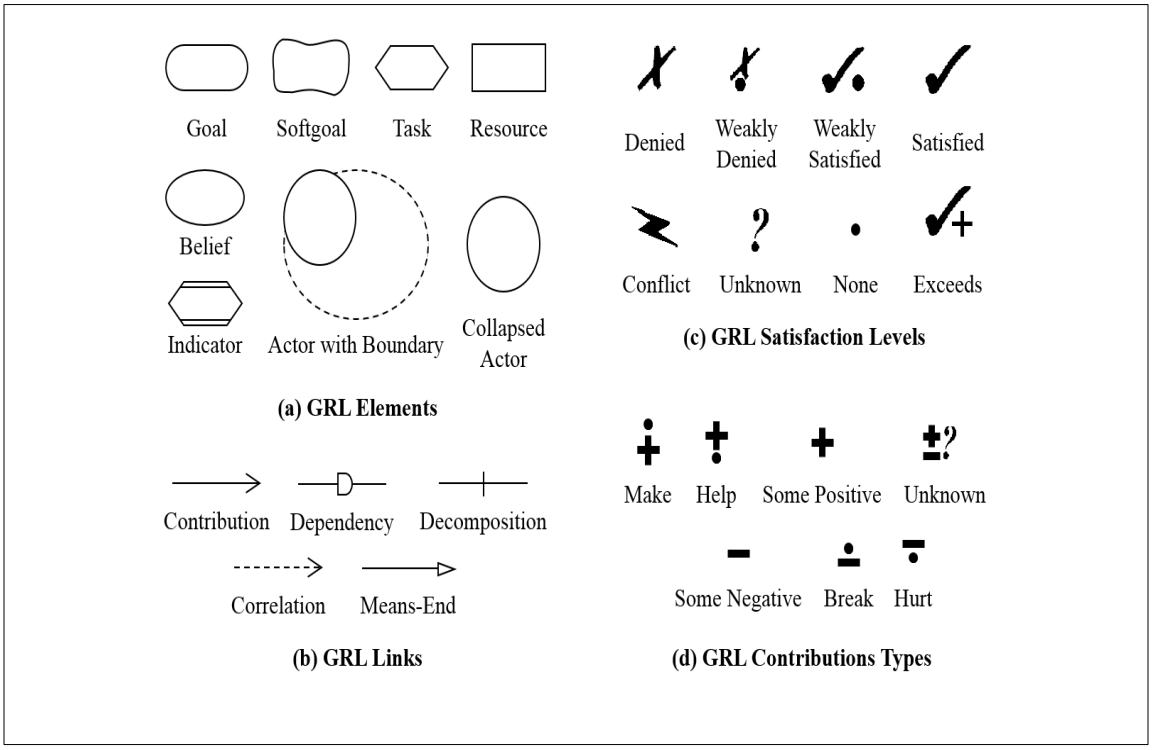


Figure 4 Summary of the GRL notation [14]

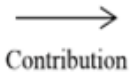
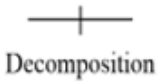
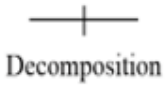
Based on Figure 4, I select *softgoals* (clouds) elements to represent NFRs and design decisions, indicating that these cannot be achieved in an absolute manner. *Tasks* (hexagons) are selected to represent patterns, tactics, the parts of a software architecture where a pattern/tactic is implemented, and software architectures, representing ways of achieving a softgoal. *An actor with boundary* (dotted circle) is used to represent an architect of a software architecture. *Solid lines* (contribution links) indicate the desired impacts of one element on another element. *Contribution types* are determined by labels. These labels

indicate various degrees of positive (+) or negative (-) contributions (see Figure 4 for the complete set of labels). *Decomposition links* allow an element to be decomposed into sub-elements [14], such as AND, IOR and XOR. I use AND decomposition links to represent the connection between a software architecture and its patterns and tactics because all the patterns are required in a software architecture before the NFRs are satisfied. I used it also to represent the connection between the parts of a software architecture and the patterns and tactics because all the patterns and tactics are needed to be implemented in a part of a software architecture. Furthermore, I use XOR to connect a general goal to its alternatives (software architectures). The satisfaction levels *Denied, Weakly Denied, Satisfied, Weakly Satisfied, Conflict, Unknown, None,* and *Exceeds* are used to represent the satisfaction level of NFRs.

Table 4 summarizes the correspondence between the GRL elements and our modeling.

Table 4 Correspondence between the GRL elements and our modeling

GRL Element	The Correspondence in Our Model	Justification
Softgoals (clouds)  	NFRs	To indicate that these cannot be achieved in an absolute manner
	Design Decisions	
Tasks (hexagons)  	Patterns	To represent ways of achieving a softgoal
	Tactic	
	Software Architectures	
	Software Architectures Parts	
An actor with boundary (dotted circle)  	Architect	To represent the actor (stakeholder) of a system

Solid lines (contribution links)  	The impact of patterns/tactics on NFRs	The desired impacts of one element on another element
Contribution values [-100, 100]	Degrees of positive or negative contributions	The strength of the impact of a pattern/tactic on an NFR
Decomposition link (AND)  	Represents the connection between a software architecture and its patterns and tactics	All the patterns are required in a software architecture before the NFRs are satisfied
	Represents the connection between the parts of a software architecture and the patterns and tactics	All the patterns and tactics are needed to be implemented in a part of a software architecture
Decomposition link (XOR)  	Represents the connection between the general goal and its alternatives (candidate software architectures)	A goal can be satisfied by choosing one of several alternatives
Satisfaction levels [0, 100]	Represent the satisfaction levels of NFRs	They show the satisfaction levels *Denied, Weakly Denied, Satisfied, Weakly Satisfied, Conflict, Unknown, None, and Exceeds* in the form of quantitative values

3.6.1. GRL Evaluation in jUCMNAv

Assigning an initial qualitative or quantitative evaluation value (e.g. a value between 0 and 100) to some of the elements in the GRL model is called a *GRL strategy*. The evaluation values would then be propagated to other high-level elements, through intentional links based on evaluation algorithms. The satisfaction values are calculated as well as the satisfaction of actors and of the entire model. Evaluation results are then indicated for each node of the goal graph with various satisfaction levels. This process is called a *GRL evaluation mechanism*. In this thesis, I use the range of [0 to 100] for the satisfaction values of the intentional elements, where (0) indicates unsatisfied or "*Denied*", while (100) indicates "*Satisfied*".

3.7. Ong et al.'s Method

I use Ong et al.'s method [98] to derive the contributions of patterns and tactics to NFRs and other rational (such as the design decisions explaining why a pattern/tactic has a positive or a negative contribution to an NFR) from their descriptions. Ong et al. [98] propose an explicit representation of the forces [58] involved in a pattern and their interrelationships. The representation interprets forces as non-functional requirements and uses the Non-Functional Requirements (NFR) Framework [58] to develop force hierarchies (based on soft-goal graphs) [98]. The force hierarchies are hierarchical graphs of forces and their contributions to each other [98].

Ong et al.'s method derives system concerns and resource constraints affected by a pattern through a close reading of the textual pattern description. They add to the description of a pattern by highlighting the core roles involved in the pattern solution and the system concerns addressed. System concerns are often expressed as phrases starting with an active verb (such as change, enable, or store).

3.8. Akhigbe et al.'s AHP Method

I use the Analytic Hierarchy Process (AHP) method [99] [100] [101] [102] to obtain the importance values of NFRs used in GRL models. Different approaches have been used in the literature to obtain importance levels and priorities of systems' concerns [101].

a) *Equal Relative Weights* [101], where all contributions targeting the same intentional element have equal weights, neglecting the fact that some contributors might be more important than others.
b) The *Analytic Hierarchy Process* (AHP) [99] [100] [101], a group decision approach used to organize and analyze complex decisions using pairwise comparisons. For more details about the AHP, please see [102].
c) *Round-Table Discussion and Consensus* [101], a focus group method where experts are assembled in a discussion setting. Groupings of related choices contained in models, are put up on a screen, and the experts are asked to discuss and assign relative weights to each choice in each grouping.
d) The *Delphi Process* [101] [103], a method used to reach consensus amongst a group of experts. Participants answer short questionnaires and provide reasons for their answers through several rounds. After each round, an anonymous summary of the responses is provided to the participants.

The interested reader can find more details about the previously mentioned methods and additional methods in [101].

I chose AHP among the previously mentioned methods because it does not require face-to-face meetings or surveys. The AHP and pairwise comparisons are also feasible because local decisions only require a few (often 2 or 3) elements even for large models.

In this thesis, I use Akhigbe et al.'s AHP method [100] to evaluate and obtain the importance values (priorities) of given NFRs in GRL models. Akhigbe [100] uses the AHP to obtain and evaluate the importance levels and priorities of requirements. According to Ahkigbe et al's [100], they are the first authors to use the AHP in GRL models. They applied the AHP's pairwise comparison technique to compare and get importance levels of set of goals in GRL models. They applied the following steps to get these importance levels.

a) Developing a pairwise comparison matrix for each business objective importance level or priority.
b) Normalizing the comparison matrix.

c) Averaging the value of each row in the matrix to get a corresponding rating.

They used the rating scales shown in Table 5 to obtain initial values for the comparison matrix.

Table 5 Rating Scale and symbols for pairwise comparisons

Symbol	Rating	Description
<<	1/5	is much less than
<	1/3	is less than
=	1	is equal to
>	3	is greater than
>>	5	is much greater than

3.9. Chapter Summary

In this chapter, I introduced background concepts that will be used in this thesis. I introduced the concepts of architectural development process, development frameworks, architecture drivers, and architectural elements. Architecture drivers include quality attributes (NFRs), functional requirements (a function of a system or components of a system, something that the system should do), context, and constraints (restrictions on any provided solution). Architecture elements include architectural patterns and architectural tactics. While architectural patterns express high-level design decisions, an architectural tactic is a design strategy that addresses a particular Quality Attribute (QA). Using patterns encourage consistency and increase the speed of development. Patterns also help users of a framework to understand and leverage the framework. A development 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.

I also introduced the tools and the methods which have been used in this thesis, such as Archie, the Goal-oriented Requirement Language (GRL), Ong's method, and Akhigbe et al.'s AHP method. These concepts and tools are described in general as well as according to the specific considerations taken by this thesis.

Chapter 4. Adaptation of Archie for Determining Patterns and Tactics Implemented in a Software Architecture

In this chapter, I present how I adapted Archie for the extraction of both architectural patterns and tactics implemented by a software architecture. After applying *Archie* to a software implementation, I validate the results, and determine the overlaps between the determined patterns and tactics. In Section 4.1, I present our motivation to adapt the *Archie* tool. Sections 4.2 illustrates our adaption process of *Archie* using an example in a particular context (big data systems). In Section 4.3, I present another illustrative example in a different context (System-of Systems (SoS)). Finally, Section 4.4 provides a summary of the chapter. A simpler version of the adaption of Archie in this chapter has been published in [104]. A more detailed version of the adaption has been published in [105].

4.1. Motivation to Adapt Archie

As discussed in Chapter 3, a manual search of the implemented patterns and tactics in system is possible, but it is not practical for large systems. Therefore, different alternative methods have been proposed in the literature to extract patterns and tactics from an implementation such as *matching method between the provided services of a framework and its patterns/tactics* [16], *pattern instantiation* (assigning the roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a practical design) [70], *matching methods between the problem statement of architecture and the applied patterns* [18]. Various tools provide some level of automated tactic or pattern detection. These tools include *Archie* [11][12][13], *Design Pattern Miner (DP Miner)* [25], *Architecture Reconstruction Plugin for Eclipse (MARPLE)-DPD* [29], *Fujaba* [33], *DP++* [34], *SPOOL* [35], and *Osprey* [36].

I chose to use *Archie*. It is an extensible tool so that additional tactics and patterns can be added by defining terms related to these tactics/patterns, with their weights. *Archie* also returns where the tactics/patterns are implemented, allowing us to validate the results

using manual inspection. It has an interactive interface and it is a plugin of the Eclipse platform. The efficiency of *Archie* for tactic detection has been demonstrated on several large systems including Hadoop [11][12][13]. *Archie* makes the extraction process faster by decreasing time and effort spent searching the patterns and tactics based on their related terms.

I noted some limitations to Archie preventing its use as it is, for our purposes. Its list of tactics needs to be supplemented to account for the range of tactics relevant to our case studies. It does not consider architectural patterns. However, because of the extensibility of *Archie*, I have been motivated to supplement its list of tactics by adding more tactics. I also added the potential for architectural patterns detection. I acknowledge that *Archie* was not designed for patterns. It was only developed to detect tactics in source code by training a classifier to determine related terms and their weights.

In this thesis, I avoid the training stage by adding terms related to tactics and patterns directly. These terms are identified from the documentation of patterns/tactics and literature. Training is not performed in this work for two reasons. First, I need a significant number of tagged samples of code segments from different systems to train a classifier on. Second, according to its creators, *Archie* would not work for patterns by training a classifier to detect terms related to patterns as it is the case for tactics [21]. The given reason is that unlike a tactic, a pattern usually consists of multiple elements forming its structure, of relationships between these elements and of a described behavior of the elements.

I are motivated to use terms from the documentation of the patterns by numerous studies [86][87][88][89] that have shown that developers tend to use meaningful terms to name variables, methods, and classes and provide meaningful comments which offer insights into the purpose of the code [12]. I recognize, however, that this adaptation could suffer from a lack of precision. This is the reason why I validate the results returned by Archie manually for our case studies.

I perform the following steps to adapt *Archie* to be used to detect architectural tactics and patterns for a given implementation. First, I determine the context/domain of the implementation. If *Archie* has not been adapted yet with tactics/patterns for that context/domain, I determine the patterns and tactics that are relevant in the context/domain and add these determined patterns and tactics to *Archie*.

4.2. Illustrative Example 1 (Cyber Fusion Center)

I illustrate the approach with a case study within the context of big data systems, in which architects are faced with the task of choosing a stream processing framework for a cyber fusion center. The selected framework's objective is to 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 analytic, such as detecting attacks and attack patterns, on the events at different levels of abstraction. Two candidate frameworks are considered here: *Apache Storm* [79] (a component in *Apache Metron* [80]) and *Apache Flink* [81].

Our objective with this example is to show one application of our approach, which is the selection among several architectures (frameworks in this example) based on the results of our evaluation approach and the provided rationale of each framework. In this section, I demonstrate the adaption process of *Archie* to determine the patterns and tactics in the context of this example (big data systems). In Chapters 5 and 6, I discuss the modeling and the evaluation results with respect to this example.

In the following, I present the steps for adapting *Archie* in the context of big data systems in general and data streaming frameworks specifically to determine the implemented patterns and tactics of Storm and Flink.

4.2.1. Determine the Context/Domain

I restrict the scope of the search by determining the context/domain of a system. I then focus on the patterns, and the tactics relevant to that specific context. For example, I determined the context of cyber fusion centers as *big data* systems in general and *data streaming* specifically.

4.2.2. Determine the Patterns and Tactics that Need to be Checked for an Architecture in a given Context

To perform this step, I conduct a literature review to find the most relevant patterns and tactics of architectures in the determined context.

In the cyber fusion center case study, I conducted a literature review to find the most relevant patterns and tactics of a big data system in general and a data streaming system in specific. Sena et al. [72] conducted a systematic mapping study that analyzes studies reporting on software architectures of big data systems. They identified a set of requirements and modules for big data systems. In [73], Sena et al. used the results of [72] to identify the commonly used architectural patterns for big data systems. These patterns are the ***Layers pattern***, the ***Pipes and Filters pattern***, the ***Broker pattern***, and the ***Shared-Repository pattern***. I supplemented Sena et al.' [72][73] with the most recent publications on big data systems. Since [72] [73] cover the publications up to 2017, I cover the recent ones from the year of 2017 until April 2019. The results are shown in Table 57 in Appendix A. I found the first six studies using the same search string as Sena's [72] [73], as shown in Table 57, while I got S7 and S8 from the references of S2. I got the studies S9-S16 using our search string (("Reference Architecture" OR "Reference Model") AND "Data Streaming System").

Table 58 in Appendix A lists studies found for tactics using a similar search string as for patterns. I summarize the list of the commonly used patterns and tactics for big data systems in Tables 6 and 8, respectively. Then, I filter those patterns and tactics such that only the ones that are commonly used in data streaming systems are considered, as shown in Tables 7 and 9 respectively. This has been done by looking just at the studies which reported on the use of patterns or tactics in data streaming systems.

Table 6 Commonly used patterns for big data systems

Patterns
Layers
Broker
Pipes and Filters
Shared-Repository
Observer/Publish-Subscriber
Blackboard

Table 7 Commonly used patterns for data streaming systems

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

Table 8 Commonly used tactics for big data systems

Tactics	
Authorization	ML algorithm optimization
Authentication	Unnecessary data removal
Audit Trail	Feature selection and extraction
Session Management	Parallel processing
Restart	Results polling and optimized notification
Exception Handling	Data cutoff
Scheduling	Alert correlation
Fraud Detection	Combining signature-based and anomaly-based detection
Risk control	Attack detection
Encryption	Algorithm selection
Role-based access control	Combining multiple detection methods
Controlled-access session	Dynamic load balancing
Mapreduce	Heartbeat
Data ingestion monitoring	Ping/Echo
Maintain multiple copies	Checkpoint
Dropped netflow detection	Load Balancing
Security data transmission	Pooling
Alert ranking	Cancel
Active Redundancy	Retry
Time Stamp	Restart
Time-out	

Table 9 Commonly used tactics for data streaming systems

Tactics	
Authorization	Maintain multiple copies
Authentication	Pooling
Audit Trail	Cancel

Tactics	
Session Management	Checkpoint
Restart	Load Balancing
Exception Handling	Active Redundancy
Scheduling	Time Stamp
Time-out	Cancel

4.2.3. Add the Determined Patterns and Tactics to Archie

The patterns and tactics related to the determined context/domain are added to *Archie*. In the context of data streaming frameworks, I add all the patterns and tactics shown in Tables 7 and 9 with terms related to these patterns and tactics. 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).

In our approach, I recognize these terms from the description of the patterns and tactics. I use sections of the description such as the name, the problem, the context, and the solution for patterns. The sources used for these descriptions include [2][53][54][56] [72] [73][84][85]. I extract from the descriptions only the most frequent and related terms to a pattern/tactic. I ignore non-related words such as 'stop' words, conjunction words, and the general words etc. The weights of our added terms are obtained by considering the frequency of each term (how many times a term is found) in the description of a pattern/tactic.

Archie searches about the related terms of a pattern/tactic everywhere in the source code (i.e. in the method names/parameters, variables names, class names, and the comments). This is because numerous studies [86][87] have shown that developers tend to use meaningful terms to name variables, methods, and classes, and to provide meaningful comments which offer insights into the purpose of the code [12] as mentioned previously in Section 4.1.

To illustrate, consider the description of the ***Broker*** pattern as documented in [84] and shown in Figure 5. I highlighted the most commonly used, the most frequent, and related terms in the description (in uppercase).

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,

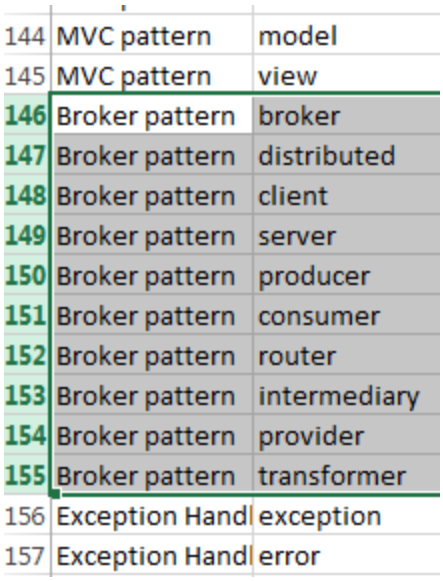
instead of focusing on low-level inter-process communication. In addition, the BROKER architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects. The BROKER pattern reduces the complexity involved in developing distributed applications because it makes distribution 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.

Figure 5 Description of the Broker pattern with the highlighted related terms

From the description in Figure 5, the most frequent related terms to the ***Broker*** pattern are *broker, client, server, distributed,* and *system*. Note that some common non-specific terms are filtered out. For instance, although *component* is one of the most frequent terms in the ***Broker*** description, I do not include it because it is a very general term (it is not related to the ***Broker*** pattern specifically).

Additional terms from [73] include *router, intermediary, provider,* and *transformer*. Additional terms determined during our manual search in the source code where a pattern/tactic is implemented (during our validation discussed in Section 4.2.4) are added to make the list more comprehensive. Examples of these terms are *producer and consumer* in the Broker's implementation, as shown in Figure 9 in Section 4.2.4. Note that I only consider single terms (not pairs or triplets of terms) in accordance with Mirakhorli's approach.

The final list of terms for the Broker pattern are *broker, distributed, system, client, server, router, intermediary, provider, producer, consumer,* and *transformer*. Figure 6 shows the Broker pattern and its related terms after I added them to the *Archie* tool.



144	MVC pattern	model
145	MVC pattern	view
146	Broker pattern	broker
147	Broker pattern	distributed
148	Broker pattern	client
149	Broker pattern	server
150	Broker pattern	producer
151	Broker pattern	consumer
152	Broker pattern	router
153	Broker pattern	intermediary
154	Broker pattern	provider
155	Broker pattern	transformer
156	Exception Handl	exception
157	Exception Handl	error

Figure 6 Broker pattern and its related terms added to the Archie tool

The remaining patterns and the tactics are added using the same process. For example, I identify the terms *"TimeStamp", "Timeout", and "Exception Handling"* as related to the availability tactics (see Appendix B for details).

Our added patterns and tactics in the context of the data streaming frameworks with their related terms are shown in Tables 10 and 11 respectively.

Table 10 Added patterns and their related terms

Context/Domain	Our Added Patterns	Related Terms
Big Data System Context/Data Streaming Framework	Layers [84]	layer, tier, responsibility, functionality
	Observer/Publish-Subscribe [84]	observe, publish, subscribe, listen, push, pull
	Pipes and Filters pattern [84]	Pipe, filter, sequence
	Broker [84]	broker, client, server, distributed system, consumer, producer, intermediary, router, provider, transformer
	Shared Repository [84]	reposit, repository, store, storage

Table 11 Added tactics and their related terms

The Related NFR	Our Added Tactics	Related Terms
Reliability/Availability	Retry [75]	Retry, reprocess, reaction
	Restart [75]	Restart, close, start, stop
	Time Stamp [75]	Timestamp, time, clock, stamp
	Timeout [75]	Time, timeout, timer
	Cancel [75]	Cancel
	Exception Handling [75]	Exception, handle, try, catch
Performance	Maintain Multiple Copies [75]	Multiple, copy, copies, maintain

4.2.4. Experimental Results with Storm and Flink

The analysis of the results of applying the adapted *Archie* on Storm and Flink is shown in Tables 12 and 13, respectively. Tables 12 and 13 show the detected and non-detected tactics/patterns for Storm and Flink and the number of their related terms. 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 occurrences of the detected terms (number of Java classes where these terms are detected), and the frequency of a pattern/tactic (number of the classes used in the implementation of a pattern/tactic). The tables also show if the tactic/pattern is detected by *Archie* or not, and the threshold where a tactic or a pattern is detected (a value calculated by Archie that determines the likelihood that a given class is associated with a tactic/pattern).

Table 12 Analysis of the results of applying Archie to Storm

#	Tactics	Number of Related Terms	Number of Detected Terms	# of Occurrences of Detected terms	Frequency of Tactics	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	4	14	14	Yes	TP	<=0.4
2	**Heartbeat**	13	2	2	2	Yes	TP	<=0.4
3	**Ping/Echo**	10	2	4	1	Yes	TP	<=0.3
4	**Exception Handling**	5	5	9	9	Yes	TP	<=0.3
5	**Authenticate**	10	1	16	4	Yes	TP	<=0.3
6	**Time Stamp**	4	1	159	9	Yes	TP	<=1
7	**Resource Pooling**	10	4	58	3	Yes	TP	<=0.6
8	**Audit Trail**	10	3	5	1	Yes	TP	<=0.7
9	**PBAC**	10	2	7	0	Yes	FP	<=0.3
10	**RBAC**	10	5	118	4	Yes	TP	<=0.5

11	**Resource Scheduling**	10	4	74	2	Yes	TP	<=0.3
12	**Session Management**	10	2	11	4	Yes	TP	<=0.2
13	**Load Balancing**	10	1	4	2	Yes	TP	<=0.4
14	**Restart**	4	1	3	3	Yes	TP	<=1
15	**Time-out**	3	1	140	1	Yes	TP	<=1
16	**Cancel**	1	1	6	4	Yes	TP	<=1
17	**Active Redundancy**	10	3	10	0	Yes	FP	<=0.1
18	**Checkpoint**	12	0	0	0	NO	NA	-
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained on Terms**	**Number of Detected Terms**	**# of Occurrences of Detected terms**	**Frequency of Patterns**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	4	4	Yes	TP	<=0.9
2	**Broker**	10	1	19	6	Yes	TP	<=1
3	**Observer/Publish-Subscribe**	4	1	3	3	Yes	TP	<=0.2
4	**Pipes and Filters**	3	1	123	15	Yes	TP	<=0.5
5	**Shared-Repository**	4	0	0	0	NO	NA	-

Table 13 Analysis of the results of applying Archie to Flink

#	Tactics	Number of Trained on terms	Number of Detected Terms	# of Occurrences of Detected terms	Frequency of Tactics	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	3	28	9	Yes	TP	<=0.6
2	**Heartbeat**	13	2	79	2	Yes	TP	<=0.4
3	**Ping/Echo**	10	1	7	2	Yes	TP	<=0.4
4	**Exception Handling**	5	4	117	16	Yes	TP	<=1
5	**Authenticate**	10	2	22	3	Yes	TP	<=0.5
6	**Time Stamp**	4	2	460	21	Yes	TP	<=1
7	**Resource Pooling**	10	1	94	5	Yes	TP	<=0.7
8	**Audit Trail**	10	1	3	0	Yes	FP	0.1
9	**PBAC**	10	2	35	0	Yes	FP	<=0.3
10	**RBAC**	10	5	56	0	Yes	FP	<=0.4
11	**Resource Scheduling**	10	2	944	4	Yes	TP	<=0.4
12	**Session Management**	10	1	131	2	Yes	TP	<=0.2
13	**Load Balancing**	10	2	10	4	Yes	TP	<=0.6
14	**Restart**	4	1	168	10	Yes	TP	<=1
15	**Time-out**	3	1	265	1	Yes	TP	<=1
16	**Cancel**	1	1	423	25	Yes	TP	<=1
17	**Active Redundancy**	10	4	16	4	Yes	FP	<=1

18	**Checkpoint**	10	1	51	3	Yes	TP	<=0.5
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained on Terms**	**Number of Detected Terms**	**# of Occurrences of Detected terms**	**Frequency of Patterns**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	24	5	Yes	TP	<=0.9
2	**Broker**	10	1	63	7	Yes	TP	<=1
3	**Observer/Publish-Subscribe**	4	2	61	8	Yes	TP	<=0.4
4	**Pipes and Filters**	3	2	254	22	Yes	TP	<=1
5	**Shared-Repository**	4	0	0	0	NO	NA	-

Figures 7 and 8 show a sample of the detected patterns and tactics when I run the Archie tool on the source code of Flink and Storm respectively.

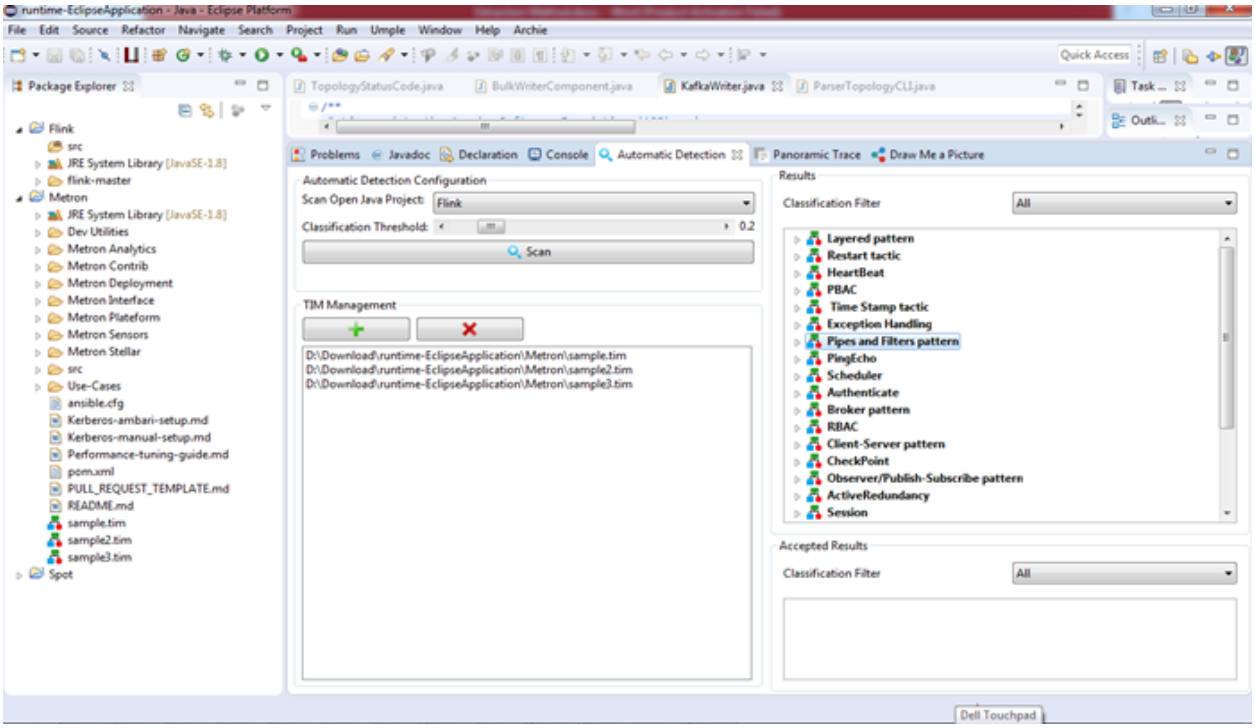


Figure 7 Sample of the detected patterns/tactics for Flink framework

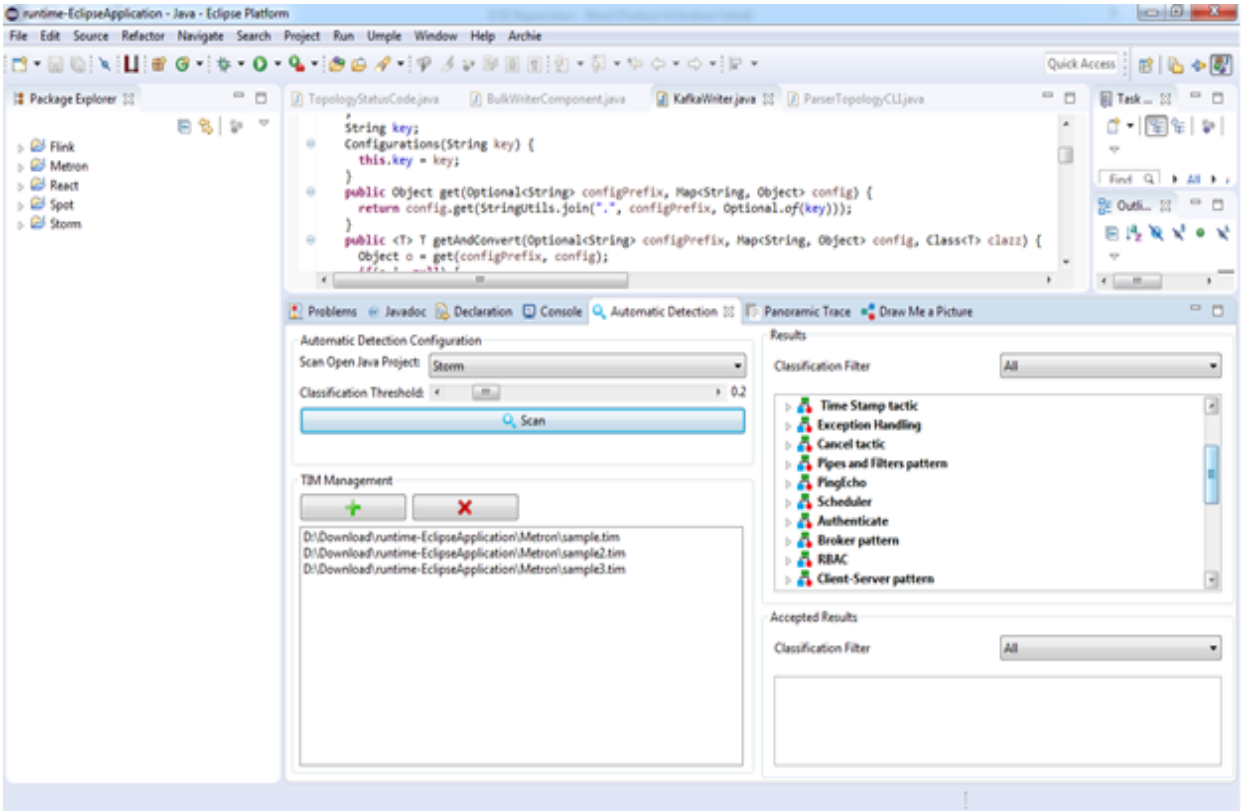


Figure 8 Sample of the detected patterns/tactics for Storm framework

I validate the results of applying *Archie* on the candidate frameworks by hunting for the occurrences of the detected patterns/tactics manually in the source code/documentation/websites of the candidate frameworks. The goal of this step is to ensure the validity of our results. Tables 84 and 85 in Appendix C detail our validation results.

I determine the numbers of the True Positives (TP) and False Positives (FP). A TP is a pattern/tactic detected by Archie that is effectively implemented, a FP is a pattern/tactic that *Archie* reports as implemented but that is found to be not effectively implemented based on our manual inspection. For instance, *Archie* detects that tactic "*Authenticate"* is implemented in Storm (TP), and this was effectively confirmed by our validation. *"RPAC"* and *"BPAC"* tactics are detected by *Archie* in Flink (FP), but they were not found to be effectively implemented based on our manual validation. Notice that I do not determine the numbers of the False Negatives (FN) nor True Negatives (TN). A (TN) being a tactic/pattern not implemented that *Archie* does not detect and a (FN) being a tactic/pattern implemented that *Archie* fails to detect. This is because in order to determine FN and TN, a very deep inspection of the code is needed. Architectural patterns/tactics can be implemented in various ways. I am able to verify TP and FP because *Archie* indicates areas of the code where the presence of a tactic/pattern is suspected. This considerably limits the amount of inspected code. Consequently, only a value for *precision* can be calculated for the detected tactics/patterns.

I calculate the value of precision in terms of the number of the True Positives (TP) and the False Positives (FP) using equation 1.

$$Precision = TP/(TP + FP) \quad (1)$$

Table 14 shows the numbers of TP, FP, and accordingly, the precision for both Storm and Flink.

Table 14 Numbers of TP, FP, and Precision for Storm and Flink

Framework	TP	FP	Precision
Storm	19	2	0.90
Flink	18	4	0.82

Figure 9 shows source code segments where the ***Broker*** pattern is detected in Storm and some of its related terms.

```
157   @Override
158   public void start() {
159     // setup Zookeeper
160     zookeeperConnectString = topologyProperties.getProperty(ZKServerComponent.ZOOKEEPER_PROPERTY);
161
162     zkClient = new ZkClient(zookeeperConnectString, 30000, 30000, ZKStringSerializer$.MODULE$);
163
164     // setup Broker
165     Properties props = TestUtilsWrapper.createBrokerConfig(0, zookeeperConnectString, brokerPort);
166     props.setProperty("zookeeper.connection.timeout.ms","1000000");
167     KafkaConfig config = new KafkaConfig(props);
168     Time mock = new MockTime();
169     kafkaServer = TestUtils.createServer(config, mock);
170
171     org.apache.log4j.Level oldLevel = UnitTestHelper.getLog4jLevel(KafkaServer.class);
172     UnitTestHelper.setLog4jLevel(KafkaServer.class, org.apache.log4j.Level.OFF);
173     // do not proceed until the broker is up
174     TestUtilsWrapper.waitUntilBrokerIsRunning(kafkaServer,"Timed out waiting for RunningAsBroker State",100000);
175
176     for(Topic topic : getTopics()) {
177       try {
178         createTopic(topic.name, topic.numPartitions, true);
179       } catch (InterruptedException e) {
180         throw new RuntimeException("Unable to create topic", e);
181       }
182     }
183     UnitTestHelper.setLog4jLevel(KafkaServer.class, oldLevel);
184     if(postStartCallback != null) {
185       postStartCallback.apply(this);
186     }
187   }
```

```
103     }
104     public KafkaComponent withBrokerPort(int brokerPort) {
105       if(brokerPort <= 0)
106       {
107         brokerPort = TestUtils.RandomPort();
108       }
109
110       this.brokerPort = brokerPort;
111       return this;
112     }
113
114     public KafkaComponent withTopics(List<Topic> topics) {
115       this.topics = topics;
116       return this;
117     }
118
119     public List<Topic> getTopics() {
120       return topics;
121     }
122
123     public int getBrokerPort() {
124       return brokerPort;
125     }
126
127
128     public String getBrokerList()  {
129       return "localhost:" + brokerPort;
130     }
131
```

```
238
239   public List<byte[]> readMessages(String topic) {
240     SimpleConsumer consumer = new SimpleConsumer("localhost", 6667, 100000, 64 * 1024, "consumer");
241     FetchRequest req = new FetchRequestBuilder()
242             .clientId("consumer")
243             .addFetch(topic, 0, 0, 100000)
244             .build();
245     FetchResponse fetchResponse = consumer.fetch(req);
246     Iterator<MessageAndOffset> results = fetchResponse.messageSet(topic, 0).iterator();
247     List<byte[]> messages = new ArrayList<>();
248     while(results.hasNext()) {
249       ByteBuffer payload = results.next().message().payload();
250       byte[] bytes = new byte[payload.limit()];
251       payload.get(bytes);
252       messages.add(bytes);
253     }
254     consumer.close();
255     return messages;
256   }
```

```
131
132   public <K,V> KafkaProducer<K, V> createProducer(Class<K> keyClass, Class<V> valueClass) {
133     return createProducer(new HashMap<>(), keyClass, valueClass);
134   }
135   public KafkaProducer<String, byte[]> createProducer()
136   {
137     return createProducer(String.class, byte[].class);
138   }
139
140   public <K,V> KafkaProducer<K,V> createProducer(Map<String, Object> properties, Class<K> keyClass, Class<V> valueClass)
141   {
142     Map<String, Object> producerConfig = new HashMap<>();
143     producerConfig.put("bootstrap.servers", getBrokerList());
144     producerConfig.put("key.serializer", "org.apache.kafka.common.serialization.ByteArraySerializer");
145     producerConfig.put("value.serializer", "org.apache.kafka.common.serialization.ByteArraySerializer");
146     producerConfig.put("request.required.acks", "-1");
147     producerConfig.put("fetch.message.max.bytes", ""+ 1024*1024*10);
148     producerConfig.put("replica.fetch.max.bytes", "" + 1024*1024*10);
149     producerConfig.put("message.max.bytes", "" + 1024*1024*10);
150     producerConfig.put("message.send.max.retries", "10");
151     producerConfig.putAll(properties);
152     KafkaProducer<K, V> ret = new KafkaProducer<>(producerConfig);
153     producersCreated.add(ret);
154     return ret;
155   }
```

Figure 9 Source code segment where Broker pattern is detected in Storm and its related terms

4.2.5. Overlaps between Patterns and Tactics

I follow Mirakhorli's approach in [21] to determine the overlaps between patterns and tactics. An overlap is declared when the intersection of the sets of classes detected for a tactic and a pattern is nonempty. I determine the overlap between patterns and tactics for two objectives. First, to show and represent the interaction between patterns and tactics in the GRL model as I will see in Chapter 6. Second, to remove duplicates; tactics

implemented both inside and outside of a pattern's implementation. We, consequently, remove the duplicate of tactics in the GRL model, as shown in Chapter 6.

Tables 15 and 16 show the results of overlap determination between tactics and patterns in Flink and Storm respectively. Column 4 shows the number of shared classes. For example, in Table 15, two classes are common to the *"Timestamp"* tactic and the ***Pipes and Filters*** pattern in Flink, while six classes are common to the *"Checkpoint"* tactic and the ***Pipes and Filter*** pattern. The *"Timeout"* and *"Exception Handling"* tactics have three classes in common with the ***Observer/Publish-Subscrib***e pattern. The ***Layers*** pattern has no overlap with any tactic, while there is a single class between the *"Cancel"* tactic and the ***Broker*** pattern. In Table 16, tactics *"Close", "Heartbeat", "Schedule", and "Timeout"* have three classes in common with the ***Pipes and Filters*** pattern in Storm. The *"Exception Handling"* tactic has three classes in common with the ***Observer/Publish-Subscribe*** pattern. The ***Layers*** pattern has no overlap with any tactic.

Table 15 Results of the overlap in the Flink framework

Pattern	Shared Java Class(s)	Tactic(s)	Overlaps
Pipes and Filter	DataStream.java TimestampITCase.java	Timestamp	2
	CoStreamCheckpointingITCase.java EventTimeWindowCheckpointingITCase.java ResumeCheckpointManuallyITCase.java StateCheckpointedITCase.java StreamCheckpointNotifierITCase.java StreamCheckpointingITCase.java	Checkpoint	6
	SchedulingStrategy.java	Resource Scheduling	1
	StreamTask.java	Close, Cancel, Exception Handling	1
	StreamExecutionEnvironment.java StreamTask.java	Timeout	2
Broker	StreamIterationHead.java	Cancel	1
Observer/Publish-Subscribe	AsynchronousFileIOChannel.java StackTraceSampleCoordinator.java PendingCheckpoint.java	Timeout	3

	PartitionRequestServerHandler.java PartitionRequestClientHandler.java AbstractReader.java	Exception Handling	3
	PendingCheckpoint.java	Cancel, timestamp	1
	KinesisDataFetcher.java PendingCheckpoint.java	checkpoint	2
Layers	-	-	-

Table 16 Results of the overlap in the Storm framework

Pattern	Java Class(s)	Tactic(s)	Overlaps
Pipes and Filter	EventHubReceiverImpl.java HBaseWindowsStore.java KafkaSpout.java	Close	3
	Nimbus.java	Checkpoint	1
	Utils.java Nimbus.java	Authentication	2
	TestStatsUtil.java ResourceAuthorizer.java AuthorizedUserFilter.java Nimbus.java	Authorization	4
	TestStatsUtil.java Nimbus.java Slot.java	Heartbeat	3
	Nimbus.java Slot.java ConstraintSolverStrategy.java	Resource Scheduling	3
	TimeOutWorkerHeartbeatsRecoveryStrategy.java Executor.java Nimbus.java	Timeout	3
Broker	KafkaSpoutTopologyMainNamedTopics.java	Retry	1
	KafkaUnit.java	Close	1
Observer/Publish-Subscribe	MqttPublisher.java MqttPublishFunction.java StormCommon.java	Exception Handling	3
	ExecutorShutdown.java	Close	1
Layers	-	-	-

4.3. Illustrative Example 2 (Healthcare Supportive System)

In this section, I present an example in a different context/domain, namely the Healthcare Supportive Home-System of Systems (HSH-SoS) architecture, published in [106].

Our objective is to show another application of our approach. In this section, I present the adaption of *Archie* to determine the patterns and tactics in the context of this example (System of Systems).

4.3.1. Patterns and Tactics Relevant to System of Systems

A literature review was conducted to find the most relevant patterns and tactics of a System of Systems (SoS) in general and a Healthcare Supportive Home-System of Systems (HSH-SoS) architecture in specific.

Garces et al. [106] conducted a study of literature to identify the architectural patterns more appropriate for architecting SoS. They identified a set of requirements and properties for SoS as well as the patterns: ***Centralized Architecture pattern, Service-Oriented Architecture (SOA), Micro-Services Architecture, Enterprise Service Bus, Publish-Subscribe, Pipes and Filters, Trickle-Up Software, Reconfiguration Control Architecture, Contract Monitor, Pace-Layers,*** and ***Evolution Styles***. Since the search in [106] was up to the year of 2018, I supplemented with most recent publications on SoS systems from January 2018 to October 2019 as shown in Table 60 in Appendix A. I used the same search string and ran queries on Scopus and Google Scholar. The search was constrained to Title-Abstract-Keywords. As a result, I found eleven primary studies that addressed patterns. I summarize the patterns relevant to the SoS architecture in Table 17.

In regard to tactics, I used the same search strings as for the patterns substituting the term "Architectural Pattern" for "Architectural Tactic" as shown in Table 61 in Appendix A. I used the data libraries Scopus and Google Scholar and limited the search from January 2010 to October 2019. I could not find any study reporting on tactics in SoSs in specific. We, therefore, searched about the tactics in the documentation and websites of SoSs. I also considered the common tactics reported by Mirakhorli [11] and Harrison [107]. The tactics considered for SoSs are presented in Table 61 in Appendix A and summarized in Table 18.

Table 17 Most patterns in SoS

Patterns
Centralized Architecture
Service-Oriented Architecture
Enterprise Service Bus
Publish-Subscribe
Pipes and Filters
Trickle-Up Software pattern
Reconfiguration Control Architecture
Contract Monitor
Pace-Layers
Reflective Architecture

Table 18 Most Tactics in SoS

Tactics
Audit Trail
Authentication
Authorization
Restart
Time Stamp
Timeout
Retry
Load Balancing
Pooling

4.3.2. Add the Determined Patterns and Tactics to Archie

The relevant tactics to the SoSs are the same as the ones for big data systems. As Archie has already been adapted with these tactics (in Table 18) in the previous case study, only the patterns in Table 17 (except for ***Publish-Subscribe*** and ***Pipes and Filter patterns***) need to be added to Archie. I determined and added terms related to these patterns from their description as in the previous case study. Our added patterns and their related terms in the context of the SoSs are shown in Table 19.

Table 19 Added Patterns and their related terms

Context/Domain	Our Added Patterns	Related Terms
System of Systems (SoS) Context/ Healthcare Domain.	Centralized Architecture [106]	Central, controller, hub, distribute
	Service-Oriented Architecture (SOA) [106]	Service, oriented, architecture, distribute, process, acti, SOA
	Enterprise Service Bus [106]	Enterprise, service, bus, connect, data transform, protocol transform, routing
	Trickle-Up Software pattern [106]	Multi, layer, trickle up, data management, aggregate
	Reconfiguration Control Architecture [106]	Dynamic, reconfiguration control architecture, degrade
	Contract Monitor [106]	Contract monitor, interface, contract

The results of applying *Archie* to the HSH-SoS and the validation are shown in section 7.2. I calculate the numbers of the True Positives (TP) and the False Positives (FP) in the HSH-SoS architecture. The numbers of the TP, FP, and accordingly the precision for HSH-SoS are shown in Table 20.

Table 20 Numbers of TP, FP, and Precision for HSH-SoS

System	TP	FP	Precision
HSH-SoS	16	4	0.80

4.4. Chapter Summary

In this chapter, I use *Archie* to extract the patterns and tactics of an architecture. *Archie* considers thirteen tactics from three quality attributes in Java-based systems. In addition to these thirteen tactics, I added seven tactics and five patterns to account for the context of the data streaming frameworks I also added six additional patterns to account for the context of SoSs. I found these tactics and patterns based on a literature search on the most commonly used patterns/tactics in the respective contexts. Then, I added terms related to the additional tactics and patterns to *Archie*. These terms are determined from the description of the tactics and patterns. As a result of applying the *Archie* tool on the source

code of an architecture, a set of patterns and tactics can be detected and presented as candidate patterns and tactics in an architecture.

Chapter 5. Modeling Patterns, Tactics, and their Contributions to NFRs using GRL

In this chapter, I model the determined patterns and tactics and their impact on Non-Functional Requirements (NFRs). The goal of this work is to create an inventory of GRL models for patterns and tactics that would be used to build GRL models for architectures (step 2 of our approach) as discussed in Chapter 6. To model the determined patterns and tactics, I first extract the NFRs and the contributions of the patterns and tactics on the NFRs from the descriptions of the patterns and tactics. I then calculate the contribution values of the patterns and tactics to the NFRs. Finally, I derive GRL models of patterns, tactics, and their contributions to NFRs from the description of each pattern/tactic.

I chose to use quantitative contributions rather than qualitative contributions of the patterns and tactics to NFRs. Unlike qualitative levels, quantitative values enable finer differentiation between the contributions of patterns/tactics to given NFRs. This allows more precision in the evaluation of architectures.

This chapter is organized as follows: Section 5.1 discusses the extraction method of the NFRs and the contributions of patterns and tactics to these NFRs. In Section 5.2, I calculate the contribution values of patterns and tactics to given NFRs. In Section 5.3, I derive the models of patterns, tactics and their contributions to the NFRs from the description of each pattern and tactic. Finally, Section 5.4 summarizes the chapter. A simpler version of the modeling approach in this chapter has been published in [104]. A more detailed version of the modeling approach has been published in [105].

5.1. Extraction of NFRs and the Contributions of the Patterns and Tactics to the NFRs from the Descriptions of the Patterns and Tactics

I extract the NFRs relevant to patterns and tactics, the contributions of the patterns and tactics on these NFRs, and the related design decisions from the documentation of the patterns/tactics. The design decisions motivate the contribution level of the patterns/tactics

on the NFRs. I consider the descriptions of the patterns as documented in [84] and [85] focusing on the consequences and solution sections, since most of the benefits and liabilities of a pattern are documented in these sections. The benefits and liabilities in the consequences section correspond respectively to the positive and negative contributions on the NFRs. The solution section summarizes most of the impacted qualities (NFRs) by a pattern and the contributions to these NFRs. Regarding tactics, I consider the descriptions of the tactics as documented in [73][75][107][108]. Documentation for all the considered patterns and tactics in this thesis is provided in Appendix B.

I apply Ong et al.'s [98] method described in Section 3.7 to extract NFRs, design decisions, and the contributions of the patterns and tactics on the NFRs. I added to the description by underlining the benefits, liabilities, the affected NFRs, and reasons for the positive or negative impact of the patterns or tactics on the NFRs. The reasons for a positive or negative impact of a pattern/tactic on a NFRs correspond to design decisions behind the application of a pattern/tactic. These design decisions are generally expressed as sentences starting with an active verb such as define, register, change, reuse, etc.

To illustrate our extraction method, consider the Broker pattern documented in [84] and shown in Figure 10. As can be noted, the NFRs affected by applying the ***Broker*** pattern appear in the consequences section. They include: *Location Transparency, Changeability, Extensibility, Portability, Interoperability, Efficiency, Reliability, Performance, and Testability.*

Figure 10 shows that the ***Broker*** pattern positively contributes to the *Location Transparency.* This is explained by the fact that the broker is responsible for locating a server by using a unique identifier such that clients do not need to know where servers are located. Likewise, servers do not care about the location of calling clients as they receive all requests from the local broker component. This is underlined to be the reason of the positive contribution of the Broker pattern to the *Location Transparency* (design decision of the ***Broker*** pattern). In contrast, the ***Broker*** pattern has lower fault tolerance *(*Negative contribution to the reliability). Indeed, when a broker fails during the execution of the program, all the applications that depend on the server or broker are unable to continue successfully. This also is underlined to be the reason of the negative contribution of the Broker pattern to the *Reliability*.

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.

Figure 10 Description of the Broker pattern with the underlined design decisions and NFRs

5.2. Calculate the Contribution Values of the Patterns and Tactics to the NFRs

In this thesis, I use quantitative contribution values in the range of [-100 to 100] for the contribution values of patterns/tactics to NFRs. Different methods may be used to determine contribution values. These methods include the *Analytical Hierarchy Process (AHP)* [99], *Delphi* [103], or *Key Performance Indicators (KPIs)* (one of the GRL notations shown in Figure 4, chapter 3). I use an alternative method by matching the strength of the relationship between a pattern/tactic and a given NFR from the literature [52][107][109][110][111][112][113] and the contribution values used in the GRL. This is motivated by our use of the range [-100 to100] for the contribution values. AHP and Delphi only work properly with the range [0 to 100]. KPIs are based on real data entered to the GRL model from which contribution values are automatically produced. I will explore the use of KPIs in future work.

Table 21 shows the correspondence between typical contribution levels found in the literature and our range [-100, 100]. If the relationship between a pattern or a tactic and a given NFR is "Key Strength", "Very Strong", or "++", then our contribution value would be "+100" in the GRL model. If the relationship is "Key Liability", "Very Weak", or "- -" our contribution value would be **(-100)**. In case that the relationship is "Neutral", "~", or "=", the contribution value would be **(0)**. If the relationship is only "Strength", "Strong" or "+", our contribution value would be any value between 1 and 99 in the GRL. Finally, the contribution value will be any value between -1 and -99 if the relationship between a pattern or a tactic and a given NFR is "Liability", "Weak", or "-".

Table 21 Match between scales from literature and the contribution values in GRL

Scale from Literature [44][48][49][50][51][52][53]	Contribution Value in GRL
Key Strength/Very Strong/++/	+100
Strength/Strong/+	[1 to 99]
Neutral/~/=	0
Liability/Weak/-	[-1 to -99]
Key Liability/Very Weak/--	-100

I determine specific contribution values between 1 and 99 or -1 and -99 based on our understanding of the strength of the contribution from the literature. For example, I calculated the contribution values of the ***Broker*** pattern to its design decisions which impact the given NFRs, as shown in Table 22. The contribution values of Broker in Table 22, express contribution values to design decisions that impact the indicated NFRs as I shown in Figure 12.

Regarding tactics, I assume that all tactics which belong to an NFR have a contribution value of **(100)**. All the tactics for a particular NFR help improving that NFR. For example, all the security tactics help improving security. Therefore, they would all have a contribution value **(100)** as shown in Table 23.

Table 22 Contribution values of the Broker pattern

	LocTra	Eff	Port	Main	Scala	Perf	Chang	Exten	Flex	Relia	Reus	Interop
Broker	100	-50	100	0	25	-6	80	100	100	-100	80	100

-LocTra: Location Transparancy – Eff: Efficiency – Port: Portability - Main: Maintainability – Scala: Scalability – Perf: Performance – Chang: Changability – Exten: Extensibility – Flex: Flexibility - Relia: Reliability – Reuse: Reusability – Interop: Interoperability

Table 23 Contribution values of the security tactics

	Kerberose	Audit Trail	RBAC/PBAC	Session Management	Authenticate
Security	100	100	100	100	100

The ***Broker*** pattern contribution value **(-6)** with *Performance* in Table 22, has been determined based on [113]. In [113], the estimated contribution value of the Broker pattern with the NFR *Performance* is **(-0.125)** in the range of [-2 to +2]. I converted this value over the range [-100 to +100] to obtain **(-6)**. As another example, the ***Broker*** pattern has a contribution value **(1.6)** in the range of [-2 to +2] with the *Changeability* NFR in [109] which corresponds to **(80)**. The ***Broker*** pattern has a contribution value **(+2)** with the NFR *Extensibility* in [109] corresponding to **(100)**. I assigned **(100)** as contribution of the ***Broker*** pattern to the *Portability* NFR since it is "Key Strength" in [108] and **(+2)** in [109]. The contribution values with *Efficiency* and *Reliability* are "Neutral" in [108] and "=" in [111] while they are negative contributions (considered as liabilities) in [84]. In [108], Harrison justifies the unknown contribution (Neutral) contribution by the fact that the

Broker pattern is a single point of failure, but this can be mitigated with duplication. In [84], the authors considered both the *Efficiency* and *Reliability* as liabilities (have negative contributions) of the ***Broker*** pattern. Based on that, I consider negative contributions with both *Efficiency* and *Reliability*. I assign **(-100)** to the contribution with the NFR *Reliability* because the ***Broker*** pattern is a single point of failure, while I assign **(-50)** with the NFR *Efficiency* because there is only some connection overhead as mentioned by Harrison in [108]. The ***Broker*** pattern has a contribution value of **(1.6)** with the NFR *Reusability* in [109]. So, I assigned **(80)** in our range. The contribution of the Broker with the NFR *Maintainability* is determined as "Strength" in [108] and "+" in [111]. However, maintainability is considered as both a benefit and a liability of the Broker pattern in [84]. This is an example of contradiction in the literature about the contribution values observed for some of the NFRs that could be due to different perspectives from the authors. I dealt with these cases by prioritizing what is documented originally in the publications with the most citations, especially well-known books. An example of a well-known book which documents several architectural patterns and is cited by several authors is the book of Buschmann et al. [84], which has been considered for all the patterns in this thesis. In accordance with [84], I assign **(0)** as a contribution value between the ***Broker*** pattern and *Maintainability*.

The contributions of the ***Broker*** pattern to the NFRs *Scalability* and *Location-Transparency* are not accurately determined in the literature. So, I determine a contribution value based on our understanding of the description of the ***Broker*** pattern in [84]. In the solution section of the pattern, *Scalability* is mentioned as one of the forces that need to be satisfied when applying the ***Broker*** pattern. Based on that, I assign **(25)** to indicate that the ***Broker*** pattern only helps to improve the NFR *Scalability*. I also assign **(100)** as contribution to the NFR *Location-Transparency* since it is considered as one of the most important and promoted benefits of the ***Broker*** pattern [84].

Using the same approach as above, I determine the contribution values of all the patterns and tactics considered in this thesis with a justification for these values in Appendix B.

5.3. Derive the Models of the Patterns and Tactics and their Contributions to the NFRs from the Descriptions of the Patterns and Tactics

I use the GRL to model patterns/tactics and their contributions to NFRs as determined in the previous section. I use the notation of the GRL shown in Figure 4 and Table 4 of Chapter 3. Figure 11 shows a general template for the modeling. First, I start with the patterns/tactics at the bottom of the model. Then, I put the NFRs at the topmost level of the model. Intermediate nodes denote design decisions leading from the patterns/tactics to the NFRs. They explain why a pattern/tactic impacts an NFR the way it does. I created an evidence table for each determined pattern/tactic for traceability (Appendix B). An evidence table records how design decisions are derived from the documentation of a pattern/tactic. Table 24 shows the evidence table of the ***Broker*** pattern.

Note that all the contribution values assigned to the contribution links connecting patterns or tactics with design decisions correspond to the contribution numbers shown in Tables 63 to 83 in Appendix B. The contribution values assigned to the links between the design decisions and NFRs at the top levels of the model, are determined based on our understanding from the descriptions of patterns/tactics. For example, in Figure 11, the contribution value between *"Pattern/Tactic"* and *"Design Decision 4"* is **(-75)**, while it is **(100)** between *"Design Decision 4"* and *"NFR 3"*. This is because, while *"Pattern/Tactic"* hurts the positive softgoal *"Design Decision 4"*, that positive design decision supports the satisfaction of *"NFR 3"*.

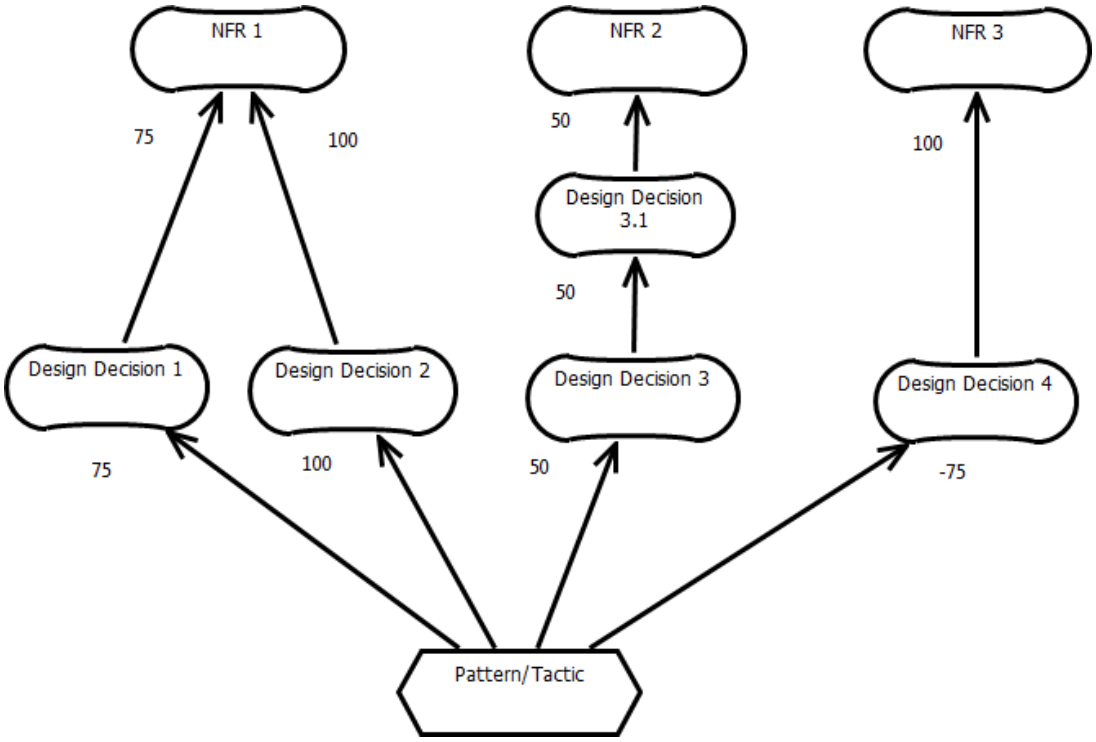


Figure 11 General model of a pattern/tactic

Table 24 Evidence table of the Broker pattern

Broker Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Location Transparency	Both clients and servers don't need to know the locations of each other by using a unique ID	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	Consequences section
Efficiency	The transmission speed between clients and servers by using indirection layers	Applications using a BROKER implementation are usually slower than applications whose COMPONENT distribution is static and known	Consequences section

		BROKER's tasks include locating the appropriate SERVER, forwarding the request to the SERVER and transmitting results and exceptions back to the CLIENT	Solution section
Portability	Low coupling services using indirection layers such as APIs, proxies and bridges	The BROKER SYSTEM hides operating system and NETWORK SYSTEM details from CLIENTS and SERVERS by using indirection layers such as APIs, proxies and bridges	Consequences section
Performance	Control the entire data flow between clients and servers	SYSTEMS that depend directly on a concrete mechanism for inter-process communication also give better performance than a BROKER architecture	Consequences section
Reliability	Single point of failure	BROKER fails during program execution. All the applications that depend on the SERVER or BROKER are unable to continue successfully	Consequences section
Reusability	Reuse existing services	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.	Consequences section
Interoperability	Transforming a specific data format in a common format when exchanging 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	Consequences section

Figure 12 shows the GRL model of the Broker pattern with its contribution to NFRs. As shown in Figure 12, the ***Broker*** pattern helps the low coupling of services by using indirection layers such as APIs, bridges, and proxies to hide the operating system and network system details from clients and servers. This promotes *Portability*. Therefore, a positive contribution of value (**100)** is shown between ***Broker*** and *Portability*. On the other hand, a broker failure during program execution causes all the applications that depend on it to be unable to continue working successfully. It is a single point of failure. Therefore, a negative contribution of type (**-100)** is shown between ***Broker*** and *Reliability*. Tested services make client applications more robust, however, testing a whole broker is tedious because of many components. This improves and, at the same time, decreases *Maintainability*. Therefore, both a positive contribution of (**50)** and a negative contribution of **(-50)** are shown between Broker and *Maintainability*. This result in *Maintainability* having an unknown contribution value (**0**).

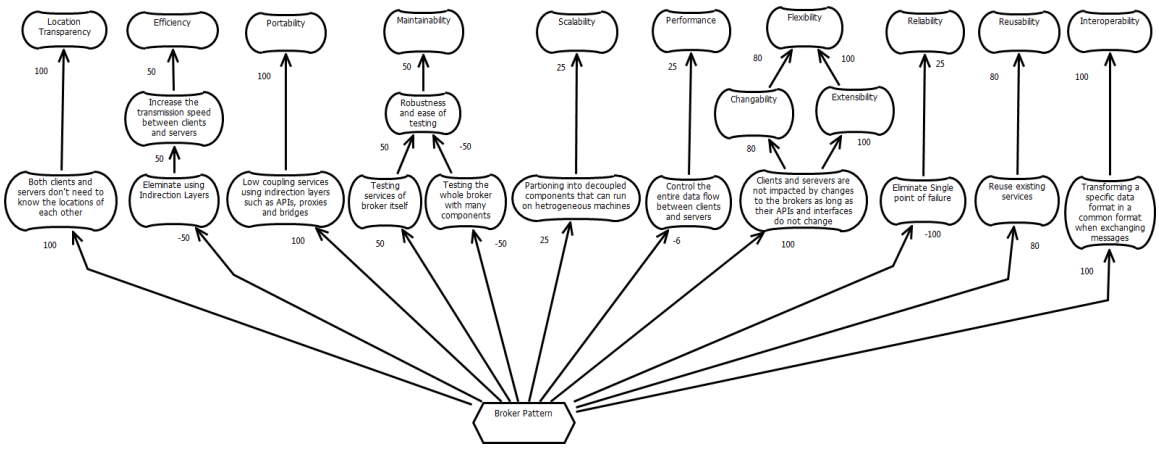


Figure 12 GRL model of the Broker Pattern

Figure 13 shows the GRL model of security tactics. All the tactics which belong to the *Security* requirement have positive contribution values. They all improve *Security*. Based on our previously stated assumption that all tactics which belong to an NFR should have equal contribution values, I assign the contribution value **(100)** to each of the tactics with *Security*. For example, the *"Audit Trail"* keeps a record of users and system actions and their effects to help trace the actions of and to identify attackers. This supports system recovery and non-repudiation. This is shown as a link with **(100)** between *"Audit Trail"*

and the *Security*. *"RBAC"* and *"PBAC"* also ensure that an authenticated user or remote computer /process has the right to access and modify either data or services. Therefore, a positive contribution of **(100)** is shown between both *"RBAC"* and *"PBAC"* and *Security*. *"Session Management", "Kerberos",* and *"Authenticate"* all help authenticate users to ensure that the user requesting a given action is the user who provided the original credentials. Therefore, a positive contribution of **(100)** is shown between each of *"Session Management", "Kerberos",* and *"Authenticate"* and *Security*.

As a result of the above, I built a GRL inventory consisting of the GRL models of the patterns and tactics relevant to our work. This inventory is to be used to build the GRL models for software architectures as discussed in chapter 6. Our GRL inventory is shown in Appendix B.

Note that, our created inventory can be used by other researchers as is aimed to be used to model architectures/systems in various contexts. Other researchers may model the same patterns and tactics in other languages such as the ones which are presented in Chapter 2 (i.e. NFR framework, i*, mathematical modeling, features models, textual techniques, etc), and investigate the requirements for these languages to support a similar approach.

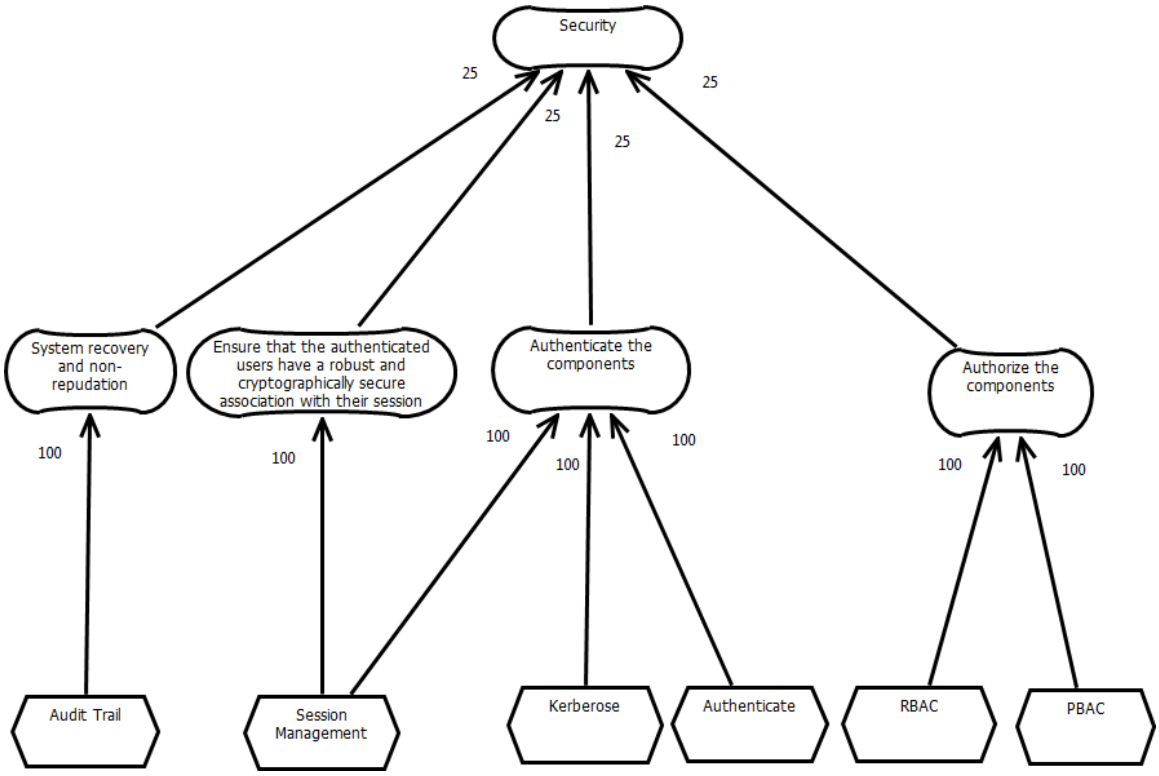


Figure 13 GRL model of the Security tactics

5.4. Chapter Summary

In this chapter, I use the Goal-oriented Requirement Language to model patterns and tactics used in an architecture. I derived the models of the patterns and tactics and their contributions to the NFRs from their descriptions. For traceability, I created an evidence table for each determined pattern/tactic. The evidence table provides an evidence of how I derive those design decisions and NFRs from the documentation of a pattern/tactic.

I then calculated the contribution values of the patterns and tactics to the NFRs based on the literature. Finally, I built a GRL inventory with all these derived GRL models of the patterns and tactics to be used to build models for architectures in Chapter 6.

Chapter 6. Modeling and Evaluating Software Architectures in terms of their Implemented Patterns and Tactics

In this chapter, I use the inventory built in chapter 5, to create GRL models for software architectures and evaluate these models with jUCMNav. I perform the following steps to model software architectures. First, I identify the most architecturally significant NFRs to be considered (discussed in Section 6.1). Then I calculate the importance values of these NFRs (discussed in Section 6.2). Furthermore, I create a model of the software architectures in terms of their patterns and tactics (discussed in Section 6.3). Finally, I evaluate the model with jUCMNav (discussed in Section 6.4). Section 6.5 summarizes the chapter. The modeling and evaluating approach of software architectures, discussed in this chapter, has been published in [105].

6.1. Determining the Relevant NFRs in Specific Contexts

In this step, I determine the most architecturally significant NFRs to be considered in the design of software architectures in a given context.

During our review (shown in Chapter 4), I searched about the NFRs relevant in specific contexts. For example, I determined the NFRs relative to the context of *big data systems* in general and to *data streaming systems* in specific. These NFRs are shown in Table 59 in Appendix A and summary is provided in Table 25. Table 26 shows the NFRs filtered to only consider the ones addressed in data streaming systems.

Table 25 Relative NFRs for big data systems

Non-Functional Requirements (NFRs)	
Scalability	Portability
Accuracy	Modularity
Performance	Consistency
Reliability	Real-time operation
Security	Availability
Usability	Maintainability
Interoperability	

Table 26 Relative NFRs for data streaming systems

Non-Functional Requirements (NFRs)	
Scalability	Security
Maintainability	Portability
Performance	Interoperability
Reliability	Availability

The NFRs considered for Storm and Flink are therefore, *Scalability, Maintainability, Performance, Portability, Availability, Reliability, Security,* and *Interoperability*.

6.2. Importance Values of Considered NFRs

Each architect (or project) has specific NFRs preferences. An architect may favor security over reliability, and another architect favor performance over the scalability and so on. To answer to the question "among the available NFRs, which is most important?", I compute the importance values of the given NFRs. I use the AHP pairwise comparison method of Akhigbe [99][100].

The steps for calculating the importance values and priorities of each NFR are as follow.

a) Obtain initial pairwise comparison.
b) Develop a pairwise comparison matrix for each NFR importance level.
c) Normalize the matrix.
d) Average the value of each row to get the corresponding rating.

I use the rating scale from Akhigbe [100] (shown in Table 5 in Chapter 3) to obtain initial values for the matrix.

For example, assuming a comparison by an architect of the importance level of the eight NFRs *Scalability, Maintainability, Performance, Portability, Availability, Reliability, Security, and Interoperability* for the data streaming frameworks Flink and Storm:

Scalability	<<	<	=	>	>>	Maintainability
Scalability	<<	<	=	>	>>	Performance
Scalability	<<	<	=	>	>>	Portability
Scalability	<<	<	=	>	>>	Availability
Scalability	<<	<	=	>	>>	Reliability
Scalability	<<	<	=	>	>>	Security
Scalability	<<	<	=	>	>>	Interoperability
Maintainability	<<	<	=	>	>>	Performance
Maintainability	<<	<	=	>	>>	Portability
Maintainability	<<	<	=	>	>>	Availability
Maintainability	<<	<	=	>	>>	Reliability
Maintainability	<<	<	=	>	>>	Security
Maintainability	<<	<	=	>	>>	Interoperability
Performance	<<	<	=	>	>>	Portability
Performance	<<	<	=	>	>>	Availability
Performance	<<	<	=	>	>>	Reliability
Performance	<<	<	=	>	>>	Security
Performance	<<	<	=	>	>>	Interoperability
Portability	<<	<	=	>	>>	Availability
Portability	<<	<	=	>	>>	Reliability
Portability	<<	<	=	>	>>	Security
Portability	<<	<	=	>	>>	Interoperability
Availability	<<	<	=	>	>>	Reliability
Availability	<<	<	=	>	>>	Security
Availability	<<	<	=	>	>>	Interoperability
Reliability	<<	<	=	>	>>	Security
Reliability	<<	<	=	>	>>	Interoperability
Security	<<	<	=	>	>>	Interoperability

Indicating that *Scalability* is less important than *Maintainability, Reliability*, and *Availability* but is much less important than *Security* and *Interoperability*. *Scalability* is

more important than *Portability* but is much important than *Performance*. *Maintainability* is less important than *Security, Portability,* and *Interoperability* but is much less than *Reliability* and *Availability* and much important than *Performance*. *Performance* is less important than *Portability* but is much less important than *Reliability, Availability, Security,* and *Interoperability*. *Portability* is less important than *Reliability, Availability, Security, and Interoperability*. *Availability*, *Reliability,* and *Security* have the same priority. However, both *Availability* and *Reliability* are less important than *Interoperability*, but *Security* is important than *Interoperability*.

These initial values are represented next in a matrix, as shown in Table 27. This matrix shows the pairwise comparison.

Table 27 Pairwise comparison matrix

	Scalability	Maintainability	Performance	Portability	Availability	Reliability	Security	Interoperability
Scalability	1	1/3	5	3	1/3	1/3	1/5	1/5
Maintainability	3	1	5	1/3	1/5	1/5	1/3	1/3
Performance	1/5	1/5	1	1/3	1/5	1/5	1/5	1/5
Portability	1/3	3	3	1	1/3	1/3	1/3	1/3
Availability	3	5	5	3	1	1	1	1/3
Reliability	3	5	5	3	1	1	1	1/3
Security	5	3	5	3	1	1	1	3
Interoperability	5	3	5	3	3	3	1/3	1

Then, the matrix is normalized.

- First, by adding up all values in each column, as shown in Table 28.

Table 28 Addition of each column

	Scalability	Maintainability	Performance	Portability	Availability	Reliability	Security	Interoperability
Scalability	1 +	0.33 +	5 +	3 +	0.33 +	0.33 +	0.2 +	0.2 +
Maintainability	3 +	1 +	5 +	0.33 +	0.2 +	0.2 +	0.33 +	0.33 +
Performance	0.2 +	0.2 +	1 +	0.33 +	0.2 +	0.2 +	0.2 +	0.2 +
Portability	0.33 +	3 +	3 +	1 +	0.33 +	0.33 +	0.33 +	0.33 +
Availability	3 +	5 +	5 +	3 +	1 +	1 +	1 +	0.33 +

Reliability	3 +	5 +	5 +	3 +	1 +	1 +	1 +	0.33 +
Security	5 +	3 +	5 +	3 +	1 +	1 +	1 +	3 +
Interoperability	5	3	5	3	3	3	0.33	1
Sum	**20.53**	**20.53**	**34**	**16.66**	**7.06**	**7.06**	**4.39**	**5.72**

- And then by dividing by the corresponding column sum, as shown in Table 29.

Table 29 Divide each value by the sum of its column

	Scalability	**Maintainability**	**Performance**	**Portability**	**Availability**	**Reliability**	**Security**	**Interoperability**
Scalability	1/20.53	0.33/20.53	5/34	3/16.66	0.33/7.06	0.33/7.06	0.2/4.39	0.2/5.72
Maintainability	3/20.53	1/20.53	5/34	0.33/16.66	0.2/7.06	0.2/7.06	0.33/4.39	0.33/5.72
Performance	0.2/20.53	0.2/20.53	1/34	0.33/16.66	0.2/7.06	0.2/7.06	0.2/4.39	0.2/5.72
Portability	0.33/20.53	3/20.53	3/34	1/16.66	0.33/7.06	0.33/7.06	0.33/4.39	0.33/5.72
Availability	3/20.53	5/20.53	5/34	3/16.66	1/7.06	1/7.06	1/4.39	0.33/5.72
Reliability	3/20.53	5/20.53	5/34	3/16.66	1/7.06	1/7.06	1/4.39	0.33/5.72
Security	5/20.53	3/20.53	5/34	3/16.66	1/7.06	1/7.06	1/4.39	3/5.72
Interoperability	5/20.53	3/20.53	5/34	3/16.66	3/7.06	3/7.06	0.33/4.39	1/5.72

- I make sure the values in each column adds up to 1, as shown in Table 30.

Table 30 New sum of each column

	Scalability	**Maintainability**	**Performance**	**Portability**	**Availability**	**Reliability**	**Security**	**Interoperability**
Scalability	0.04	0.02	0.15	0.18	0.05	0.05	0.05	0.03
Maintainability	0.15	0.04	0.15	0.02	0.03	0.03	0.07	0.06
Performance	0.01	0.01	0.02	0.02	0.03	0.03	0.05	0.03
Portability	0.02	0.15	0.08	0.06	0.05	0.05	0.07	0.06
Availability	0.15	0.24	0.15	0.18	0.14	0.14	0.23	0.06
Reliability	0.15	0.24	0.15	0.18	0.14	0.14	0.23	0.06

Security	0.24	0.15	0.15	0.18	0.14	0.14	0.23	0.52
Interoperability	0.24	0.15	0.15	0.18	0.42	0.42	0.07	0.18
Sum	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Finally, I get the average of each row to obtain the corresponding rating (in this thesis, I use a ranking of 0 to 100 for GRL importance levels, so I multiply each row average by 100), as shown in Table 31.

Table 31 Calculated the importance level of each NFR relative to the other

	Scalability	**Maintainability**	**Performance**	**Portability**	**Availability**	**Reliability**	**Security**	**Interoperability**	**Row Average**	**X 100**
Scalability	0.04	0.02	0.15	0.18	0.05	0.05	0.05	0.03	0.07	7
Maintainability	0.15	0.04	0.15	0.02	0.03	0.03	0.07	0.06	0.07	7
Performance	0.01	0.01	0.02	0.02	0.03	0.03	0.05	0.03	0.03	3
Portability	0.02	0.15	0.08	0.06	0.05	0.05	0.07	0.06	0.07	7
Availability	0.15	0.24	0.15	0.18	0.14	0.14	0.23	0.06	0.16	16
Reliability	0.15	0.24	0.15	0.18	0.14	0.14	0.23	0.06	0.16	16
Security	0.24	0.15	0.15	0.18	0.14	0.14	0.23	0.52	0.22	22
Interoperability	0.24	0.15	0.15	0.18	0.42	0.42	0.07	0.18	0.22	22
Sum	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**100**

Figure 14 below shows the calculated level of importance of the eight NFRs.

Figure 14 Calculated importance levels of the NFRs

6.3. Modeling Software Architectures in Terms of their Implemented Patterns and Tactics

A template of software architectures GRL model in terms of their implemented patterns and tactics is shown in Figure 15. I start with the evaluated software architectures and their parts at the bottom level of the model, connected with all their implemented patterns and tactics. The parts of a software architecture show where its patterns and tactics are implemented. However, adding these parts does not impact the satisfaction levels of the NFRs in the model of a software architecture. I added them so the interested reader knows where these patterns/tactics are found in the source code of an architecture. I link patterns to their implemented tactics indicating the overlaps between patterns and tactics, as discussed in the previous step. The resulting GRL model specifies that the design decisions provide an explanation of why a pattern/tactic impacts an NFR the way it does.

As shown in Figure 15, on the top of the NFRs is a more abstract (general) goal, which needs to be satisfied by these NFRs for a system. That goal can be solved by the evaluated software architectures. The general goal is connected to the architectures at the bottom of the model. The design decisions provide an explanation of why a pattern/tactic impacts an NFR the way it does at the top of the model. They push or pull the software architecture toward, or away from NFRs (e.g., better security or greater availability). For example, *"Design Decision 2"* and *"Design Decision 2.1"* pull *"Software Architecture"* 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 a reduction of the satisfaction of the *"NFR 2"* of *"Software Architecture"*. At the same time, *"Tactic 2"* which is implemented in *"Pattern 2"* helps to push the *"Software Architecture"* towards *"NFR 2"*. This is because *"Tactic 2"* positively impacts *"Design Decision 4"*, resulting in increasing the satisfaction of the *"NFR 2"*. While, *"Tactic 1"*, which is implemented in *"Pattern 1"*, hurts *"Design Decision 3"*, resulting in pushing the *"Software Architecture"* away from *"NFR 1"*.

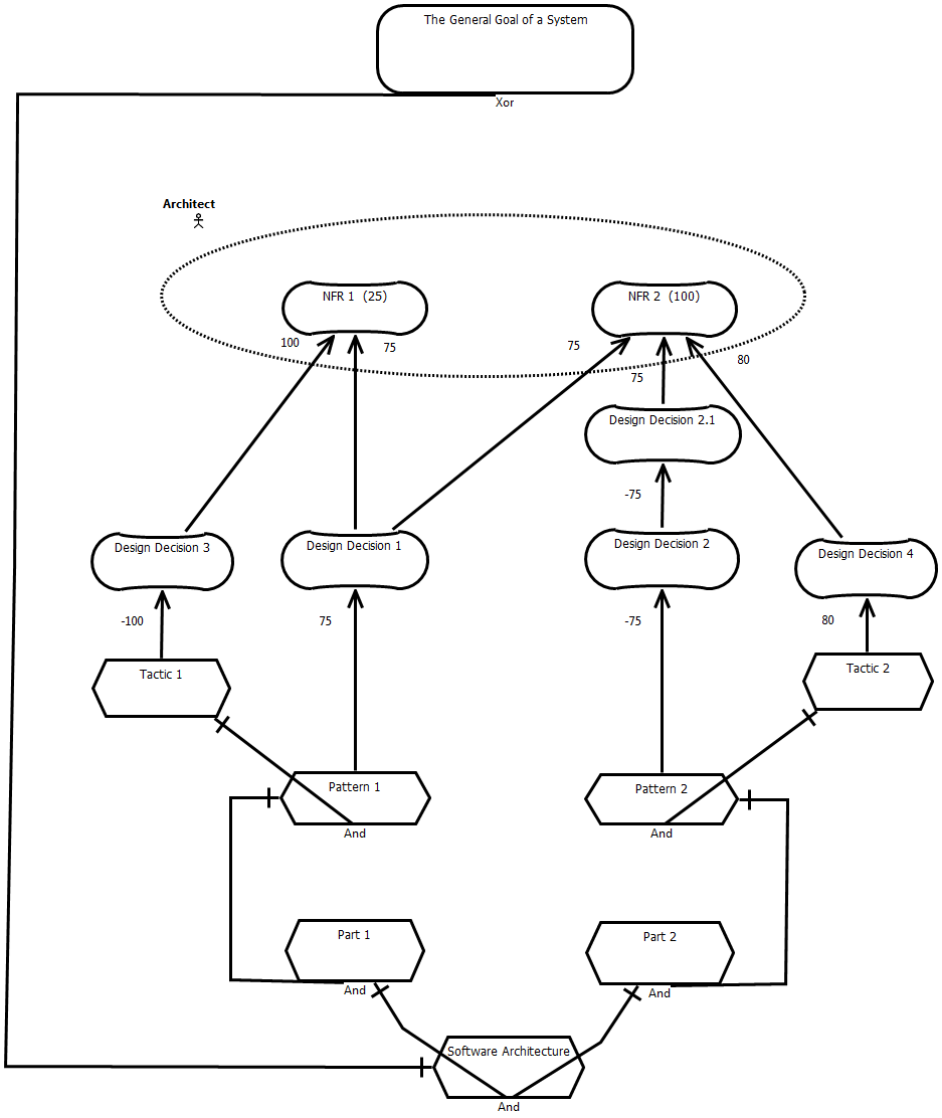


Figure 15 General GRL model of a software architecture

I use the algorithm in Figure 16 to model software architectures in terms of their detected patterns and tactics according to the general GRL model in Figure 15. This algorithm takes as *input* a list of tactics and patterns found by *Archie* in the source code, a list of the relevant NFRs for the architecture, and the inventory of the models for tactics and patterns built using the approach in Chapter 5. The *output* of this algorithm is the GRL model for the software architecture. This algorithm uses the models in the inventory as reusable GRL building blocks. It connects all the models of the determined patterns and tactics with the considered NFRs. It also considers the overlap between patterns and tactics

by connecting together all the tactics and patterns which have at least one common class in their implementations.

Algorithm Software Architecture Model Creation

Inputs: Tactics_List: List, Patterns_List: List *// Two lists of patterns and tactics found in the source code of the architecture*

SoftwareArchitecture_NFRs_List: List *// List of the NFRs of the architecture*

Inventory: Map <Pattern/Tactic, GRL_model> *//Inventory of the GRL models of the determined patterns and tactics*

Output: SoftwareArchitecture_Model: GRL *// GRL model for an architecture in terms of its implemented patterns and tactics*

// Reuse the inventory to get GRL models of the determined patterns/tactics of a Software architecture

1. **foreach** aPattern in Patterns_List do {
2. aPatternModel := Inventory.get (aPattern) *// Get GRL model of the determined pattern*
3. aPatternModels_List. add (aPatternModel) *// Add the retrieved pattern model to list of models*
4. **foreach** anNFR in aPatternModels_List {
5. **foreach** softwareArchitecture_NFR in softwareArchitecture_NFRs_List do {
6. **if** anNFR == softwareArchitecture_NFR *// if the NFR in the pattern model the same as the NFR of the software architecture*
7. **then**
8. aPatternPaths_List:=getModelPath(aPatternModels_List, SoftwareArchitecure_NFR) *// This function gets the model path which includes the pattern and its link(s) to its design decisions up to the NFR*
9. **foreach** aPatternPath in aPatternPaths_List {

```
10.        linkSoftwareArchitectureToPath(aPatternPath)   // This function links the software
       architecture to the paths of patterns which are linked to the software architecture's NFRs
11.            } // end foreach
12.        } // end foreach
13.     } // end foreach
14.   } // end foreach

15.     foreach aTactic in Tactics_List do {
16.        aTacticModel := Inventory.get (aTactic)          // Get GRL model of the determined
                                                                          tactic
17.        aTacticModels_List. add (aTacticModel)        // Add the retrieved tactic model to
                                                                       list of models
18.     foreach anNFR in aTacticModels_List {
19.        foreach softwareArchitecture_NFR in softwareArchitecture_NFRs_List do {
20.           if anNFR == softwareArchitecture_NFR    // if the NFR in the tactic model the same
                                                          as the NFR of the software architecture
21.           then
22.                aTacticPaths_List:=getModelPath(aTacticModels_List,
softwareArchitecure_NFR) // This function gets the model path which includes the tactic and
its link(s) to its design decisions up to the NFR
23.     foreach aTacticPath in aTacticPaths_List {
24.        foreach aPattern in Patterns_List do {
25.           if overlap (aPattern, aTactic) == True        // if there is an overlap between aPattern
                                                                    and aTactic
26.           then connectPatternToTactic(aPattern, aTacticPath)  //This function connects
together the overlapped patterns and tactics
27.           else linkSoftwareArchitectureToPath(aTacticPath)   // This function links the
software architecture to the paths of tactics which are linked to the software architecture's
NFRs
28.              } // end foreach
29.          } // end foreach
30.      } // end foreach
31.   } // end foreach
```

32. } *// end foreach*

33. **return** SoftwareArchitecture_Model

Figure 16 Software Architecture Model Creation algorithm

To illustrate the algorithm in Figure 16, I apply it to model the Storm framework in terms of availability and reliability. First, all the GRL models of the determined patterns, by Archie, for Storm (i.e. ***Layers, Broker, Observer/Publish-Subscriber,*** and ***Pipes and Filters***) would be retrieved from the inventory by applying the lines from 1 to 3 of the algorithm. The paths linking the detected patterns to the availability and reliability NFRs would be then retrieved from the inventory (lines 4 to 8) as shown in Figure 17. As can be noticed in Figure 17, only the models of the patterns ***Broker, Observer/Publish-Subscribe,*** and ***Pipes and Filters*** consider reliability and availability. The retrieved paths connecting the patterns to reliability and availability are highlighted using arrows. All of these paths would be then linked to the Storm framework (lines 9 and 10) through its parts. This shows where these patterns are implemented as illustrated in Figure 18. The same process is applied to the tactics (lines 15 to 22). For sake of readability, I only consider the "*Heartbeat", "Checkpoint", and "Ping/Echo"* tactics. The highlighted paths for those tactics are shown in Figure 19. Function *overlap* would be applied to check for overlap between patterns and tactics. If a tactic has an overlap with a pattern (i.e. has at least one shared class with a pattern), then the retrieved tactic's path would be linked to the overlapped pattern, otherwise, it would be linked to the Storm framework (lines 23 to 27) as shown in Figure 20 (the final model of Strom). For example, the two tactics *"Heartbeat"* and *"Checkpoint"* overlap with ***Pipes and Filters*** pattern. They are therefore, linked to the ***Pipes and Filters*** pattern. *"Ping/Echo"* has no overlap with any of the patterns, therefore, it is directly linked to the Storm framework.

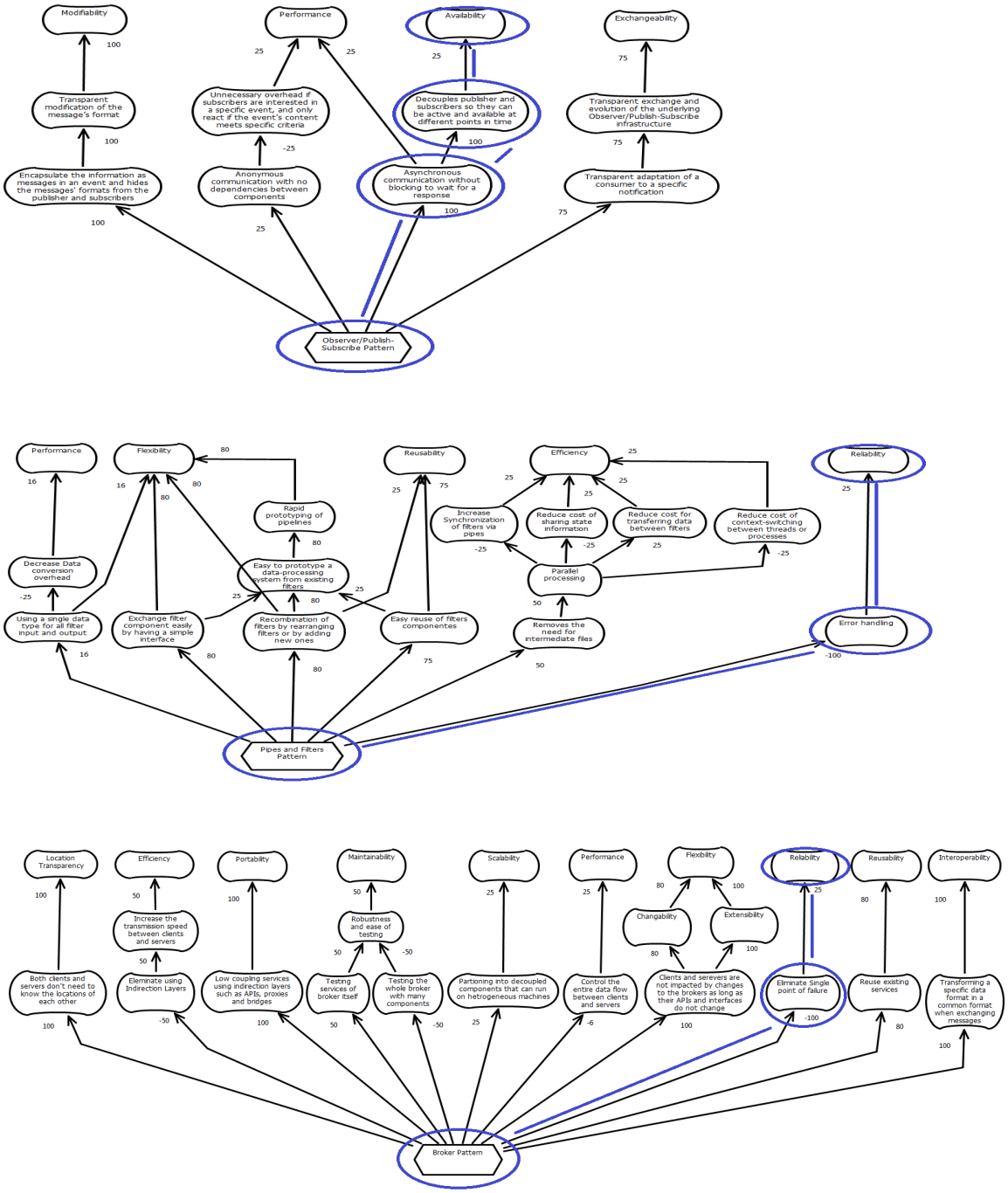


Figure 17 Paths linking the patterns Observer/Publish-Subscribe, Pipes and Filters, and Broker to the availability and reliability NFRs

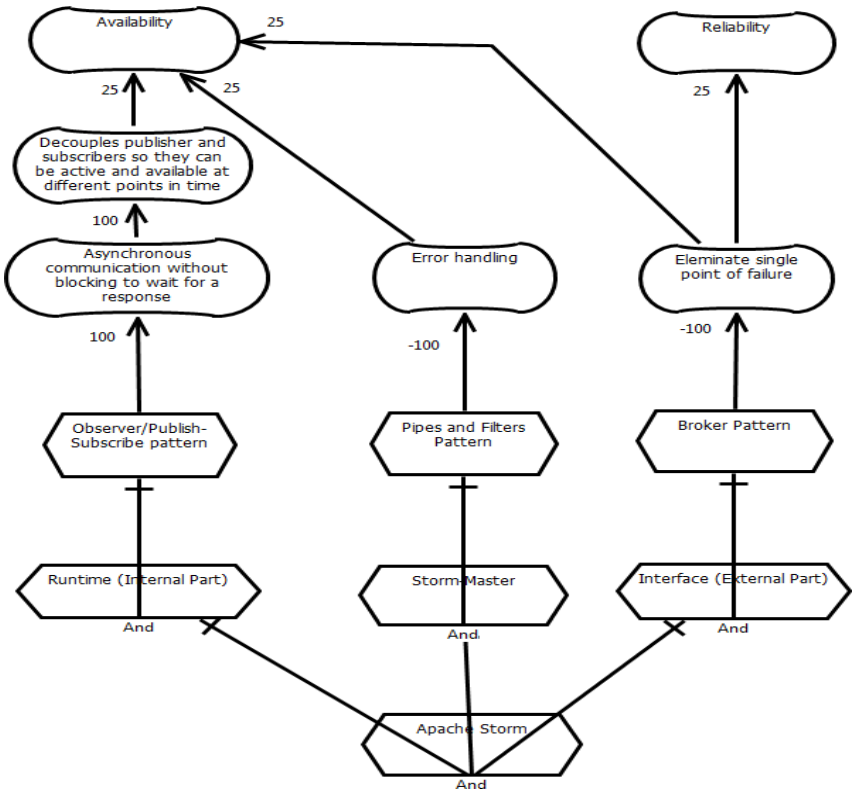


Figure 18 Linking the highlighted paths of the patterns to the Storm framework

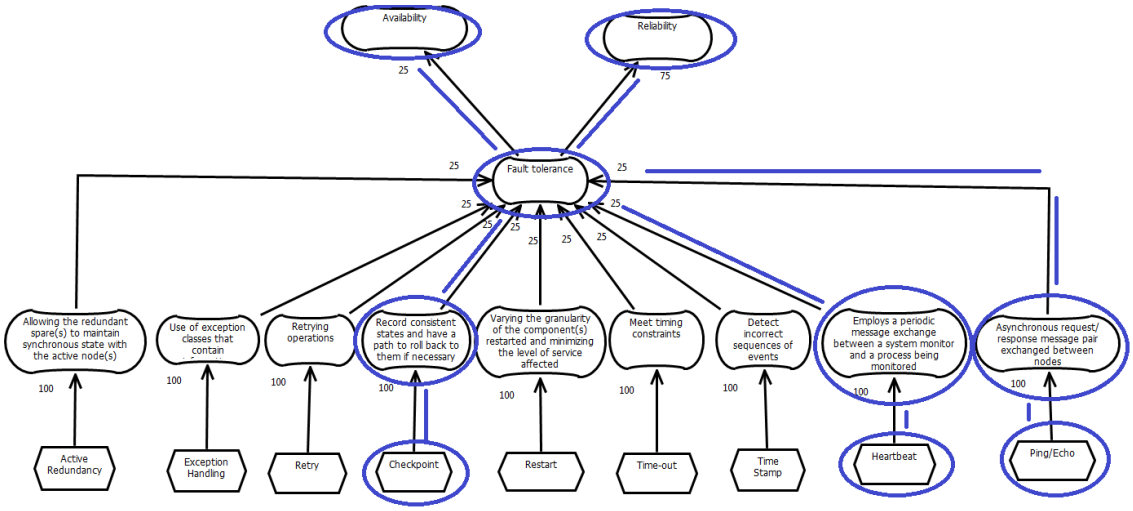


Figure 19 Paths linking the tactics Checkpoint, Heartbeat, and Ping/Echo to the availability and reliability NFRs

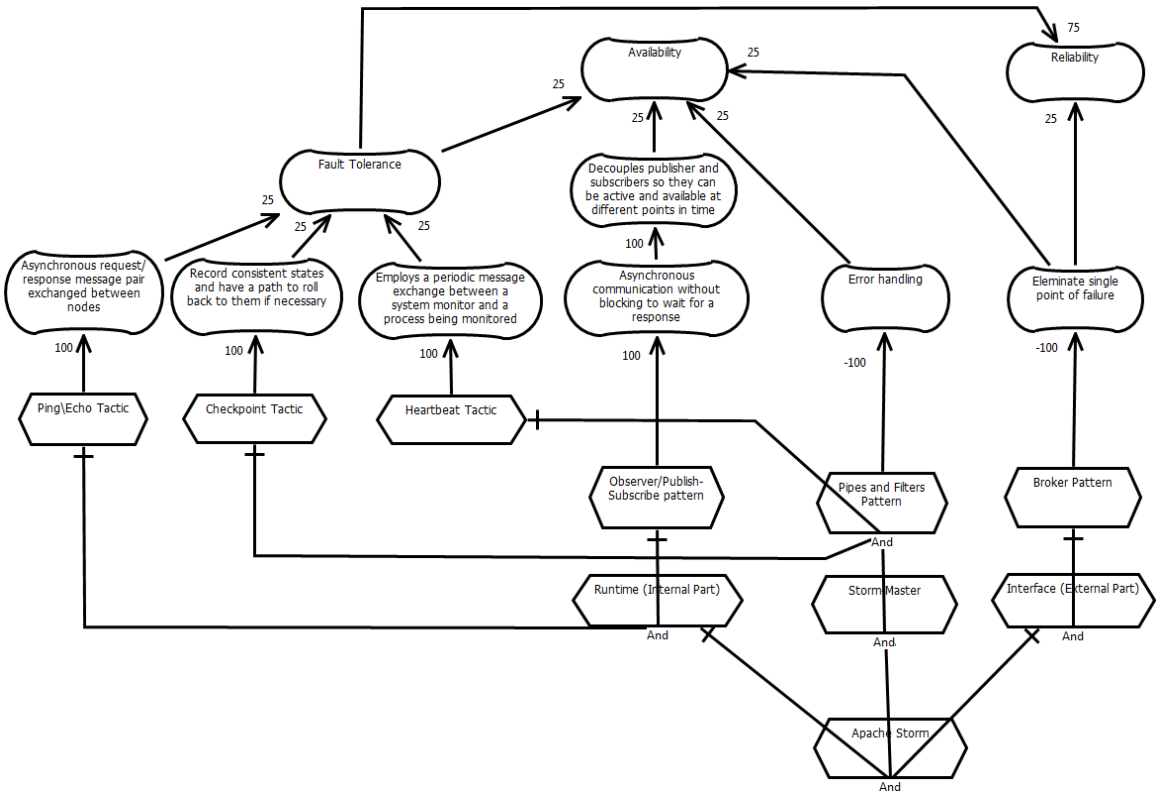


Figure 20 Final model of the Storm framework in terms of availability and reliability

Figure 21 shows the two frameworks Apache Storm and Apache Flink modeled following the general model.

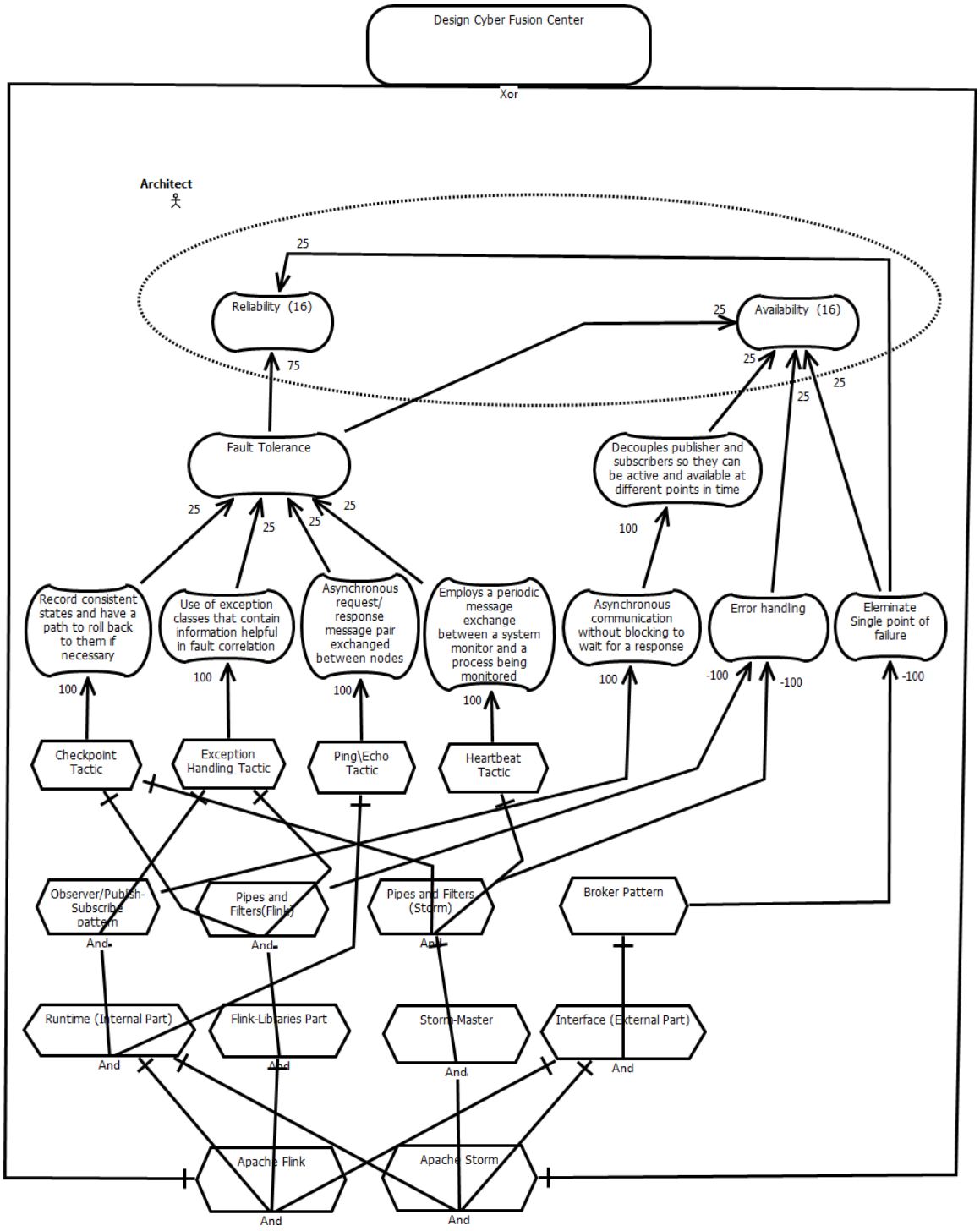


Figure 21 GRL model of Flink and Storm in terms of the Reliability and Availability requirements

Both Flink and Storm are connected to their implemented patterns and tactics. All of these patterns and tactics are directly connected to the parts of the frameworks where they are implemented. However, some of these tactics are already implemented in patterns. These tactics are linked to a pattern rather than a part of the frameworks. For example, the *"Heartbeat"* tactic is implemented in the ***Pipes and Filters*** pattern in the Storm framework. It is therefore, linked directly to the ***Pipes and Filters*** pattern. ***Pipes and Filters*** has a negative contribution of **(-100)** to the *"Error handling"* decision, which negatively impact the *Availability* requirement. This is because of the impossibility of, for example, restarting a pipeline or ignoring an error. However, the model also shows that all the tactics which belong to the *Reliability* requirement have positive contribution values to the reliability.

6.4. Using the Model to Evaluate Software Architectures

6.4.1. Evaluate the Model of the Software Architectures

I evaluate software architectures model by setting a GRL strategy, which consists of a set of intentional elements (patterns and tactics) that are given initial satisfaction values. I assign a satisfaction level of 100 (Satisfied) to a tactic or a pattern if a framework uses that tactic or pattern. The satisfaction level is otherwise assigned to be 0 (Denied). The initial values are marked with a star (*)[2] on the evaluation model. The importance values of the NFRs, inside actor (architect) boundaries, are determined following the process discussed in section 6.2. These importance values are specified between parentheses in the intentional elements (NFRs) as an integer between (0) and (100).

Figures 22 and 23 show two different strategies before running the jUCMNav propagation and getting the satisfaction levels. The first strategy in Figure 22 models the application of Apache Storm. All its implemented patterns and tactics are therefore, initially assigned to be 100 (Satisfied) and marked using (*). Figure 23 models the application of Apache Flink with all its implemented patterns and tactics initially assigned

2 (*): In the GRL, the Star (*) means an element in the GRL model is initialized manually by users not automatically by GRL/jUCMNav.

100 (Satisfied) and marked using (*). The importance values of the NFRs inside the architect boundary are then specified following the calculations discussed in section 6.2.

After the initial assignment of satisfaction levels to the tactics and patterns of an architecture, I evaluate the GRL model by running the evaluation strategies with the jUCMNav tool. This automatically propagates the initial values to the other intentional elements through their links and, therefore, calculates the satisfaction levels of each node in the GRL model. The evaluation results are indicated for each node of the goal graph with various satisfaction levels. I use the range of [0 to 100] for the satisfaction levels of NFRs. Color-coding is used to highlight what is satisfied and what is denied. The 'Green' colour indicates that an element is satisfied, the 'Yellow' colour indicates that the element is neutral, and the 'Red' colour indicates that the element is denied.

Figures 24 and 25 show the evaluation strategies, for both Storm and Flink respectively, after running the jUCMNav propagation and getting the satisfaction levels for each node in the model.

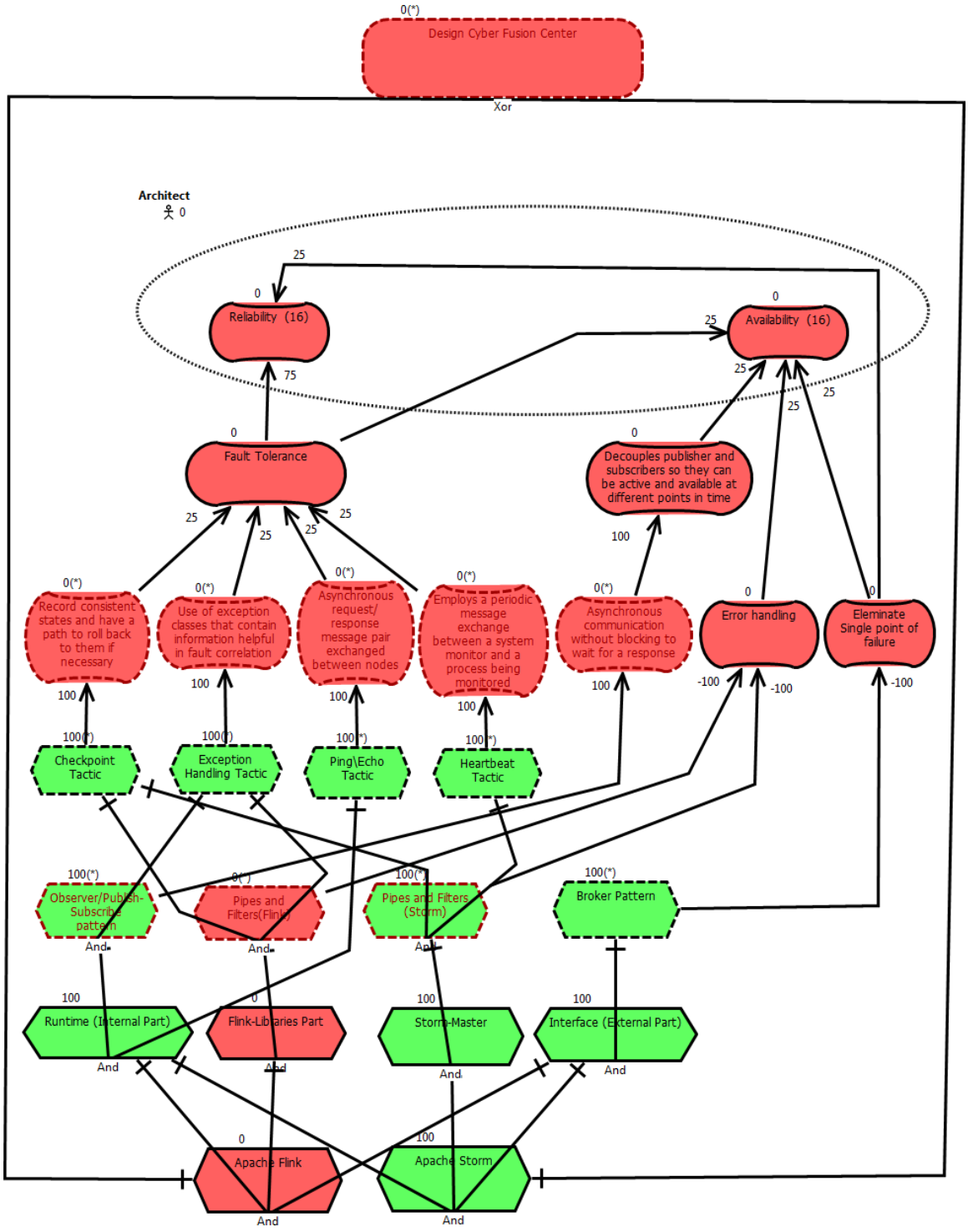


Figure 22 GRL model where patterns/tactics of Storm are initially satisfied in terms of Availability and Reliability before running the jUCMNav propagation

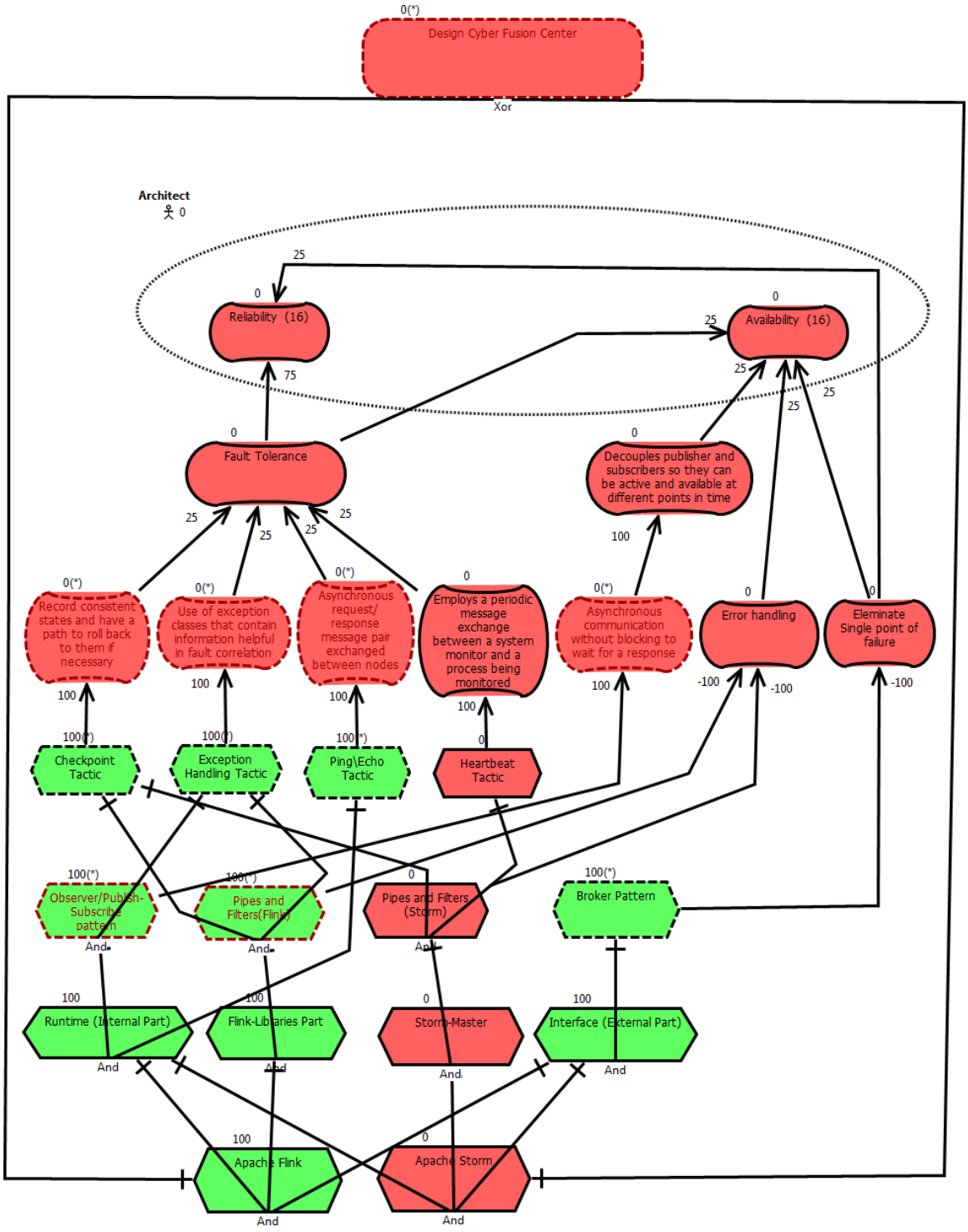


Figure 23 GRL model where patterns/tactics of Flink are initially satisfied in terms of Availability and Reliability before running the jUCMNav propagation

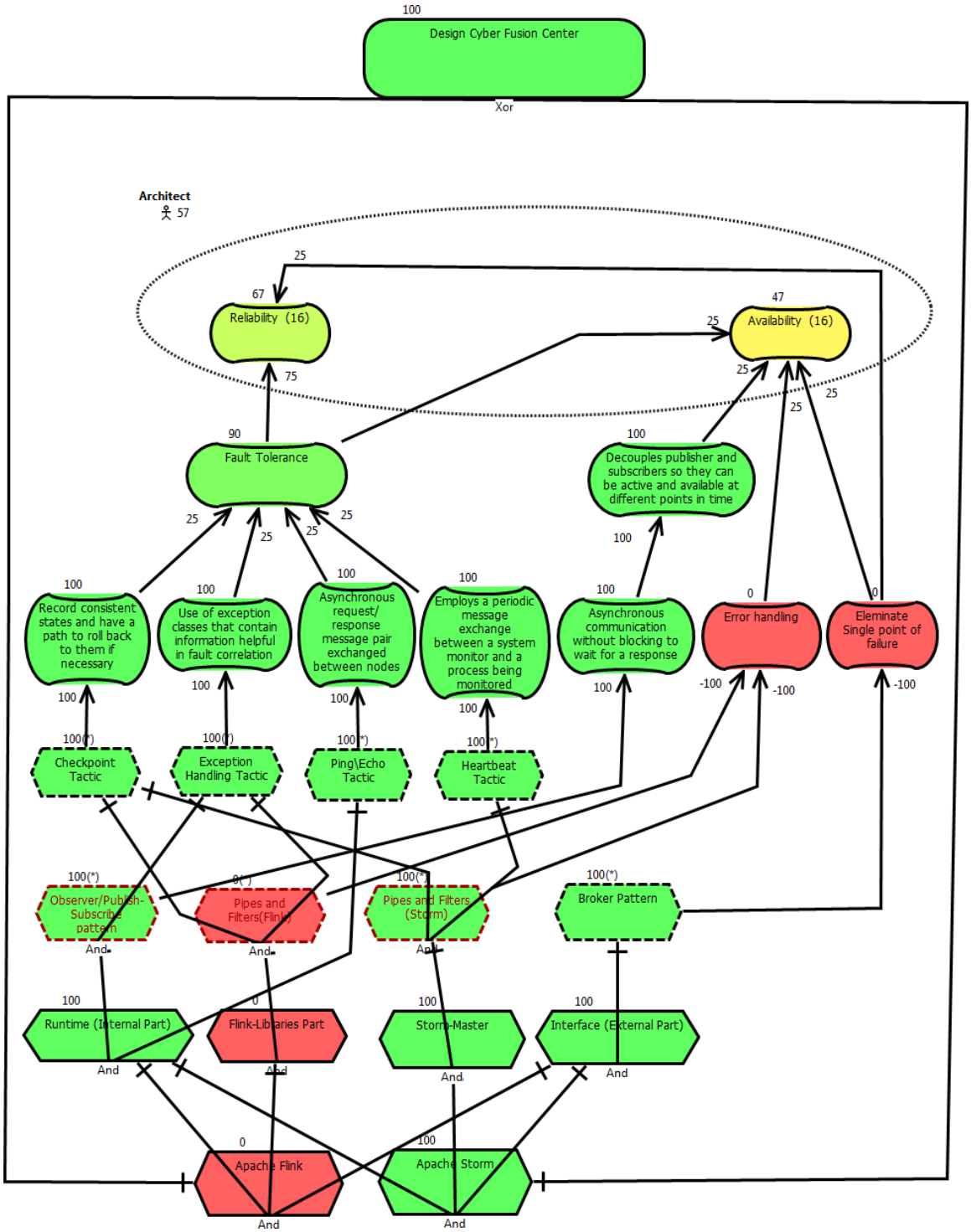


Figure 24 Strategy 1: Evaluated GRL model where patterns/tactics of Storm are initially satisfied in terms of Availability and Reliability after running the jUCMNav propagation

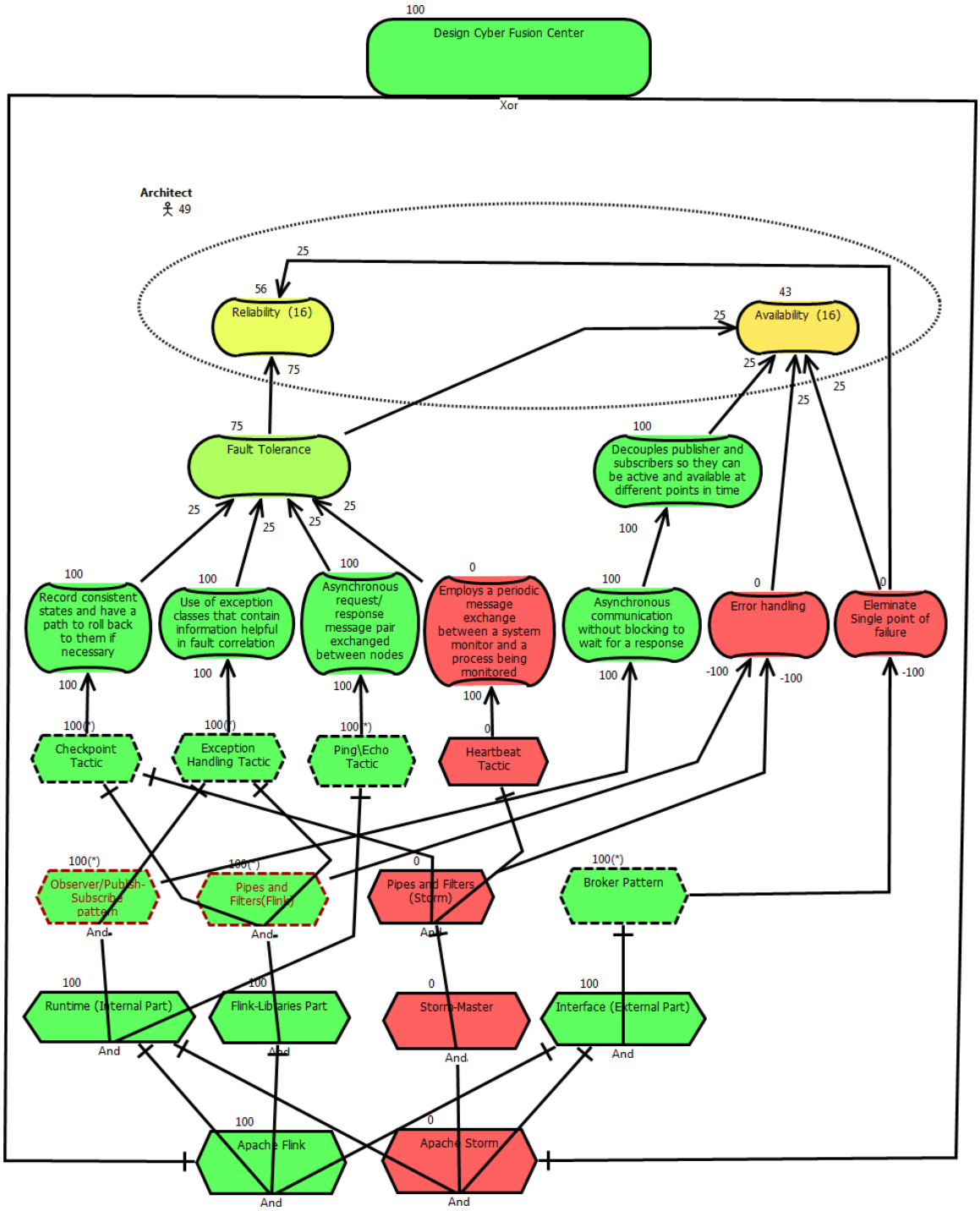


Figure 25 Strategy 2: Evaluated GRL model where patterns/tactics of Flink are initially satisfied in terms of Availability and Reliability after running the jUCMNav propagation

6.4.2. Evaluate the Software Architectures

In this last step, I evaluate the candidate frameworks based on their implemented patterns and tactics considering the importance values of the NFRs calculated in Section 6.2.

As I can see in Figures 24 and 25, the architect is more satisfied in Storm than Flink, as shown by the satisfaction value for Flink **(49)**, compared to the one for Storm **(57)**. The Storm framework has a higher satisfaction level for *Reliability* and *Availability*, which are respectively **(67)** and **(47)** compared to the Flink framework. This can be explained by the implementation of the four tactics: "*Checkpoint", "Exception Handling", "Heartbeat",* and "*Ping/Echo"*. These tactics all improve fault tolerance, which in turn improves reliability. The ***Observer/Publish-Subscribe*** pattern used in both frameworks provides asynchronous communication between components without blocking for a response. This helps decouple publishers and subscribers so that they can be active and available at different points in time, resulting in improving the availability of the frameworks. According to the above evaluation, if an architect favours *Reliability* and *Availability* over the other requirements, I would recommend Storm. More details about evaluations in terms of multiple requirements are discussed in Chapter 7.

6.5. Chapter Summary

In this chapter, I use the GRL inventory built in Chapter 5 to create the GRL model for software architectures. I first identified the most architecturally significant NFRs to be considered in the model. I then calculated the importance values of the identified NFRs. I then determined the overlap between patterns and tactics. I create a model of a software architecture by using the Framework Model Creation algorithm. This algorithm links a software architecture to its implemented patterns and tactics and connects together the overlapped patterns and tactics (which have at least one overlapped class in their implementation). I evaluate the resulting GRL model of software architectures by applying different strategies in the GRL using the range of [0 to 100] to the satisfaction levels of NFRs. The evaluation considers the importance values of given quality requirements. These important values express the preferences of stakeholders including architects.

Chapter 7. Case Studies

In this chapter, I illustrate our approach with case studies from different contexts. Three case studies are considered. The first case study is based on an industrial big data project. The objective is to assist in the task of choosing a stream processing framework for a cyber fusion center. Three candidate frameworks were short-listed by the industrial partners: Apache Storm, Apache Flink, and Apache Spark. Apache Storm and Flink have already been discussed in the previous chapters in the context of this case study. The second case study is based on a Health supportive Home –System of Systems (HSH-SoS) within the context of System of Systems (SoS). The objective is to analyze and understand the implementation and design of the HSH-SoS architecture in terms of its implemented patterns and tactics and their impact on its quality attributes. The third case study evaluates two build-automation tools, Gradle and Maven. This case study provides some validation to the main hypothesis of our research by comparing our inferred quality attributes (i.e. performance) to benchmark values.

7.1. Case Study 1: Cyber Fusion Center

This case study is about the task of choosing a stream processing framework for a cyber fusion center. The framework is to 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).

I already illustrated steps of the approach with Apache Storm and Flink in the previous chapters. A third framework, Apache Spark [82], was considered in the case study.

In the following, I introduce the frameworks Apache Storm, Apache Flink, and Apache Spark.

7.1.1. Apache Storm

Apache Storm [79] is a free, open-source, distributed real-time computation system. It can be used with various programming languages. Apache Storm consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation as needed. Figure 26 shows the architecture of Apache Storm. The interested reader may refer to [79] for more details.

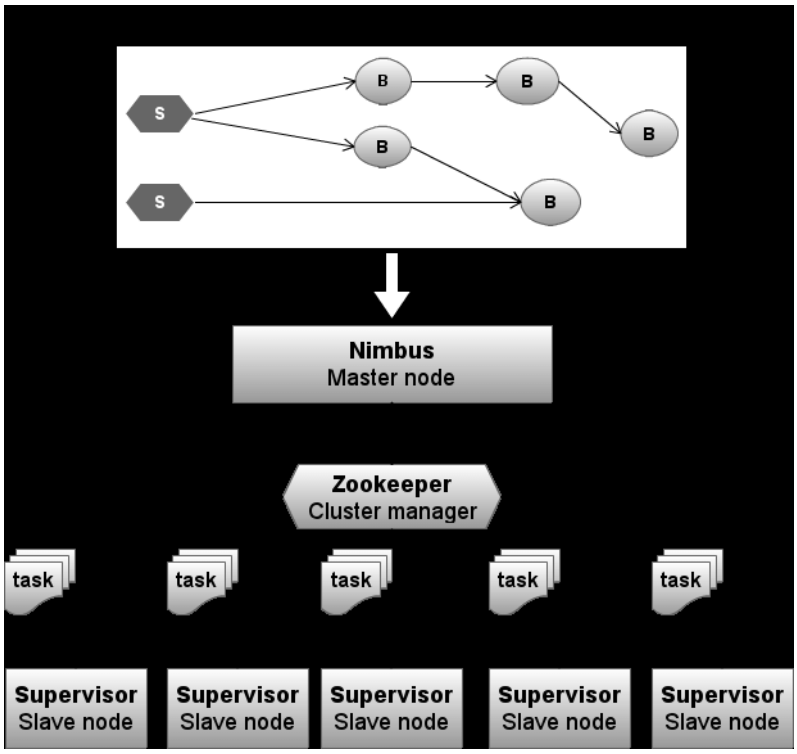


Figure 26 Apache Storm architecture. Adapted from [114]

7.1.2. Apache Flink

Apache Flink [81] is an open-source framework for distributed stream processing. It uses machine learning for applications such as threat detection, investigation, and remediation. Flink is stateful, fault-tolerant, and performs at large scale. It can run over thousands of nodes with excellent throughput and latency characteristics. Figure 27 shows the architecture of Apache Flink. The interested reader may refer to [81] for more details.

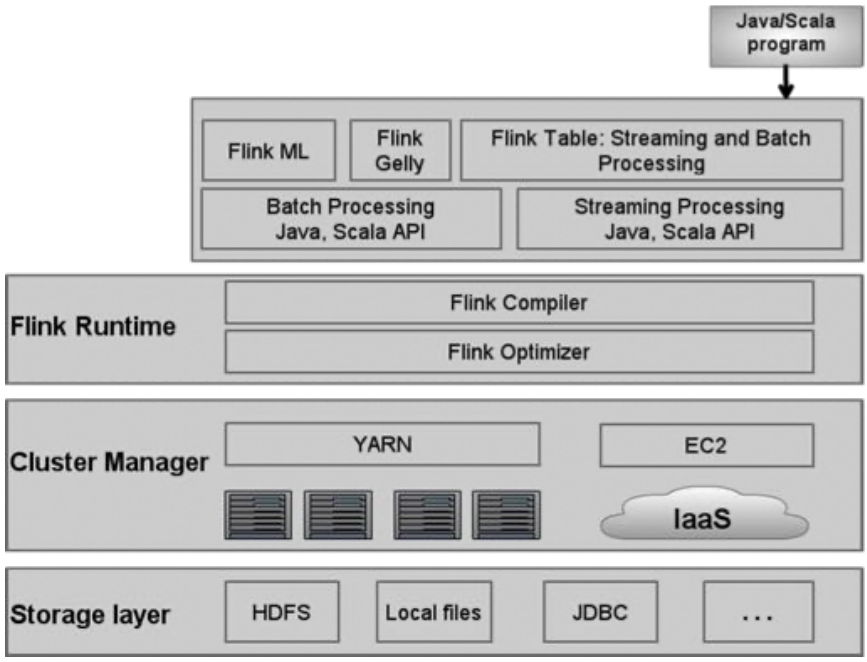


Figure 27 Apache Flink architecture. Adapted from [114]

7.1.3. Apache Spark

Apache Spark [82] is an open-source framework for distributed and large-scale data processing. It provides an interface for entire programming clusters with implicit data parallelism and fault tolerance [82]. It is a fast and general-purpose cluster computing system. Apache Spark provides distributed task dispatching, scheduling, and basic I/O functionalities. Figure 28 shows the architecture of Apache Spark. The interested reader can refer to [82] for more details.

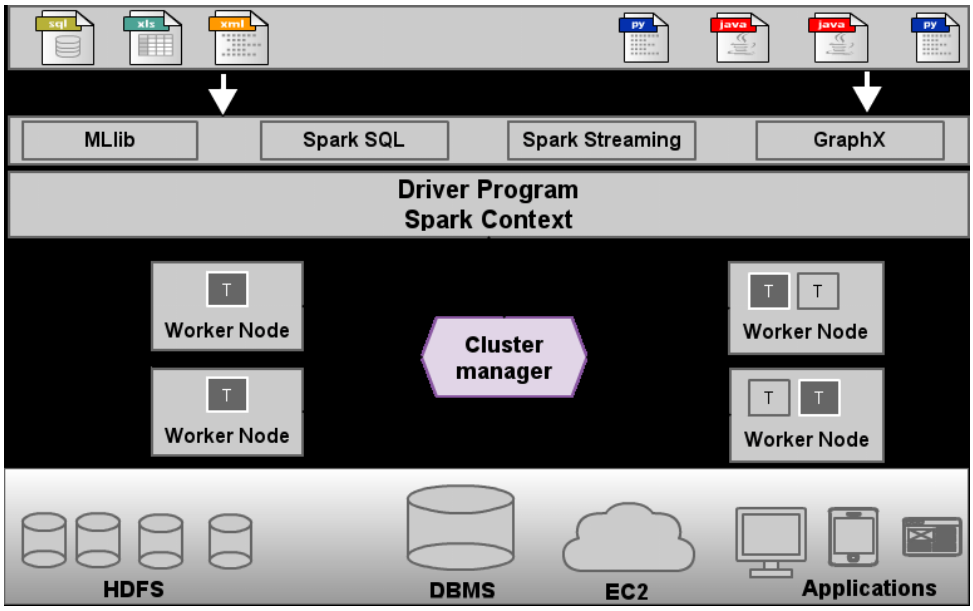


Figure 28 Apache Spark architecture. Adapted from [114]

In the next sections, I model Apache Spark and compare it to Storm and Flink. This case study shows that the approach is applicable to more than two frameworks from the same context.

7.1.4. Determining the Patterns and Tactics Implemented in a Software Architecture

I have determined the patterns and tactics of Storm and Flink in Chapter 4. In this Section, I only determine the patterns and tactics of the Spark framework.

7.1.4.1. Experimental Results with Spark

I applied the Archie tool on the source code of Apache Spark. An analysis of the results is shown in Table 32.

Table 32 Analysis of applying Archie to the Spark

#	Tactics	Number of Trained on Terms	Number of Detected Terms	Frequency of Detected Terms	Frequency of Tactic	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	5	16	8	Yes	TP	<=0.9
2	**Heartbeat**	13	1	3	3	Yes	TP	<=0.4
3	**Ping/Echo**	10	1	3	0	Yes	FP	<=0.4

4	**Exception Handling**	5	4	19	4	Yes	TP	<=0.3
5	**Authenticate**	10	1	46	13	Yes	TP	<=0.5
6	**Time Stamp**	4	2	141	17	Yes	TP	<=0.2
7	**Resource Pooling**	10	3	20	5	Yes	TP	<=0.7
8	**Audit Trail**	10	0	0	0	NO	NA	-
9	**PBAC**	10	1	3	0	Yes	FP	<=0.3
10	**RBAC**	10	1	1	0	Yes	FP	<=0.2
11	**Resource Scheduling**	10	2	13	1	Yes	TP	<=0.3
12	**Session Management**	10	2	32	8	Yes	TP	<=0.7
13	**Load Balancing**	10	0	0	0	NO	NA	-
14	**Restart**	4	3	83	9	Yes	TP	<=0.9
15	**Time-Out**	3	2	143	11	Yes	TP	<=1
16	**Cancel**	1	1	22	5	Yes	TP	<=1
17	**Active Redundancy**	10	1	4	2	Yes	TP	<=0.4
18	**Checkpoint**	12	1	14	0	Yes	FP	<=0.5
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained on Terms**	**Number of Detected Terms**	**Frequency of Detected Terms**	**Frequency of Patterns**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	11	1	Yes	TP	<=0.8
2	**Broker**	10	3	4	3	Yes	TP	<=0.3
3	**Observer/Publish-Subscribe**	4	0	0	0	NO	NA	-
4	**Pipes and Filters**	3	2	48	4	Yes	TP	<=0.4
5	**Shared-Repository**	4	2	6	2	Yes	TP	<=0.3

Figure 29 shows a sample of the detected patterns and tactics when I run *Archie* on the source code of Spark.

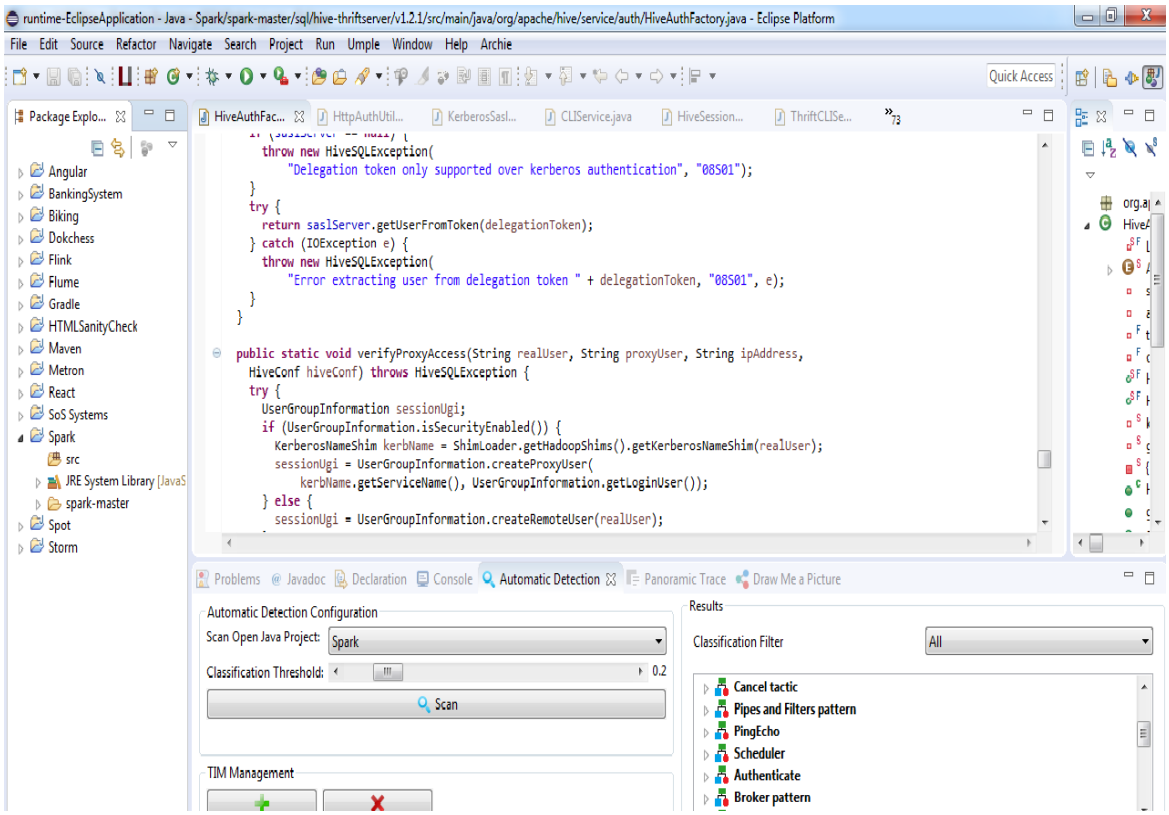


Figure 29 Sample of the detected patterns/tactics for Spark

I validated the results obtained with Spark by manually hunting for the occurrences of the patterns/tactics detected by Archie, in the source code/documentation/websites of Spark. For example, Figure 30 shows the source code where the *"Authenticate"* tactic is detected in Spark and its related terms.

```
public class AuthIntegrationSuite {

  private AuthTestCtx ctx;

  @After
  public void cleanUp() throws Exception {
    if (ctx != null) {
      ctx.close();
    }
    ctx = null;
  }

  @Test
  public void testNewAuth() throws Exception {
    ctx = new AuthTestCtx();
    ctx.createServer("secret");
    ctx.createClient("secret");

    ByteBuffer reply = ctx.client.sendRpcSync(JavaUtils.stringToBytes("Ping"), 5000);
    assertEquals("Pong", JavaUtils.bytesToString(reply));
    assertTrue(ctx.authRpcHandler.doDelegate);
    assertFalse(ctx.authRpcHandler.delegate instanceof SaslRpcHandler);
  }

  @Test
  public void testAuthFailure() throws Exception {
    ctx = new AuthTestCtx();
    ctx.createServer("server");

    try {
```

```
  @Test
  public void testSaslServerFallback() throws Exception {
    ctx = new AuthTestCtx();
    ctx.createServer("secret", true);
    ctx.createClient("secret", false);

    ByteBuffer reply = ctx.client.sendRpcSync(JavaUtils.stringToBytes("Ping"), 5000);
    assertEquals("Pong", JavaUtils.bytesToString(reply));
  }

  @Test
  public void testSaslClientFallback() throws Exception {
    ctx = new AuthTestCtx();
    ctx.createServer("secret", false);
    ctx.createClient("secret", true);

    ByteBuffer reply = ctx.client.sendRpcSync(JavaUtils.stringToBytes("Ping"), 5000);
    assertEquals("Pong", JavaUtils.bytesToString(reply));
  }

  @Test
  public void testAuthReplay() throws Exception {
    // This test covers the case where an attacker replays a challenge message sniffed from the
    // network, but doesn't know the actual secret. The server should close the connection as
    // soon as a message is sent after authentication is performed. This is emulated by removing
    // the client encryption handler after authentication.
    ctx = new AuthTestCtx();
    ctx.createServer("secret");
    ctx.createClient("secret");
```

```
private class AuthTestCtx {

  private final String appId = "testAppId";
  private final TransportConf conf;
  private final TransportContext ctx;

  TransportClient client;
  TransportServer server;
  volatile Channel serverChannel;
  volatile AuthRpcHandler authRpcHandler;

  AuthTestCtx() throws Exception {
    this(new RpcHandler() {
      @Override
      public void receive(
          TransportClient client,
          ByteBuffer message,
          RpcResponseCallback callback) {
        assertEquals("Ping", JavaUtils.bytesToString(message));
        callback.onSuccess(JavaUtils.stringToBytes("Pong"));
      }

      @Override
      public StreamManager getStreamManager() {
        return null;
      }
    });
  }

  AuthTestCtx(RpcHandler rpcHandler) throws Exception {
    Map<String, String> testConf = ImmutableMap.of("spark.network.crypto.enabled", "true");
```

Figure 30 Source code where Authenticate tactic is detected in Spark and its related terms

The numbers of True Positives (TP), False Positives (FP), and accordingly the precision for Spark together with Storm and Flink are shown in Table 33. The adapted *Archie* precision for identifying patterns and tactics is 0.90 for Storm, 0.82 for Flink, and 0.79 for Spark.

Table 33 Numbers of TP, FP, and Precision for Storm, Flink, and Spark

Framework	TP	FP	Precision
Storm	19	2	0.90
Flink	18	4	0.82
Spark	15	4	0.79

7.1.4.2. Overlaps between Patterns and Tactics

This is discussed for Flink and Storm in Section 5.4.1. Table 34 shows the overlaps between patterns and tactics in the Spark framework.

Table 34 Results of the overlaps in the Spark framework

Pattern	Shared Java Class(s)	Tactic(s)	Overlaps
Pipes and Filter	JavaAPISuite.java JavaRecoverableNetworkWordCount.java JavaAPISuite.java	Heartbeat	3
	ExternalShuffleBlockResolver.java	Close, Cancel	1
Broker	JavaStructuredKerberizedKafkaWordCount.java	Session Management	1
	JavaDirectKafkaWordCount.java	Restart	1
Observer/Publish-Subscribe	-	-	-
Layers	TransportClient.java TransportChannelHandler.java ExternalBlockHandler.java	Timeout	3
	StreamManager.java	Authorization	1
	SpecificParquetRecordReaderBase.java	Close	1

As I can in Table 34, the *"Heartbeat"* tactic has three classes overlap with the ***Pipes and Filters*** pattern. *"Close"* and *"Cancel"* has a single class overlap with the ***Pipes and Filters*** pattern. *"Session Management"* and *"Restart"* also has a single class overlap with the ***Broker*** pattern. The ***Observer/Publish-Subscribe*** pattern has no overlap with any tactic. The *"Timeout"* has three classes overlap and *"Authorization",* and *"Close"* tactics have a single class overlap with the ***Layers*** pattern.

7.1.5. Modelling Software Architectures in terms of their Implemented Patterns and Tactics

This step includes the following sub-steps:

a. Identify the most architecturally significant NFRs to be considered in the design of software architectures in the determined context.

b. Calculate the importance values of the considered NFRs according to stakeholders' preferences.

c. Model software architectures in terms of their implemented patterns and tactics.

7.1.5.1. Identify the Most Architecturally Significant NFRs to be Considered in the Design of Software Architectures in the Determined Context

This has already been done for the context of big data systems and data streaming frameworks when I applied our approach to Apache Storm and Flink in the previous chapter (see Section 6.1). The set of NFRs considered for Storm, Flink, and Spark is *Scalability, Maintainability, Performance, Portability, Testability, Availability, Reliability, Security,* and *Interoperability*.

7.1.5.2. Calculate the Importance Values of the Considered NFRs

This also has already been done for the context of big data systems and data streaming frameworks when I applied our approach to Apache Storm and Flink in the previous chapter (See Section 6.2).

7.1.5.3. Model Software Architectures in terms of their Implemented Patterns and Tactics

I modeled the three candidate frameworks Apache Storm, Apache Flink, and Apache Spark in terms of their implemented patterns and tactics, as shown in Figure 31. The model considers the connections between the patterns and tactics. For the sake of readability, I show in Appendix B, separate models based on NFRs. However, I only define here two models for Storm, Flink, and Spark. One model shown in Figure 31, in terms of five NFRs *Testability, Security, Reliability, Availability,* and *Scalability* and the other model in Figure 32, in terms of the NFR *Performance*.

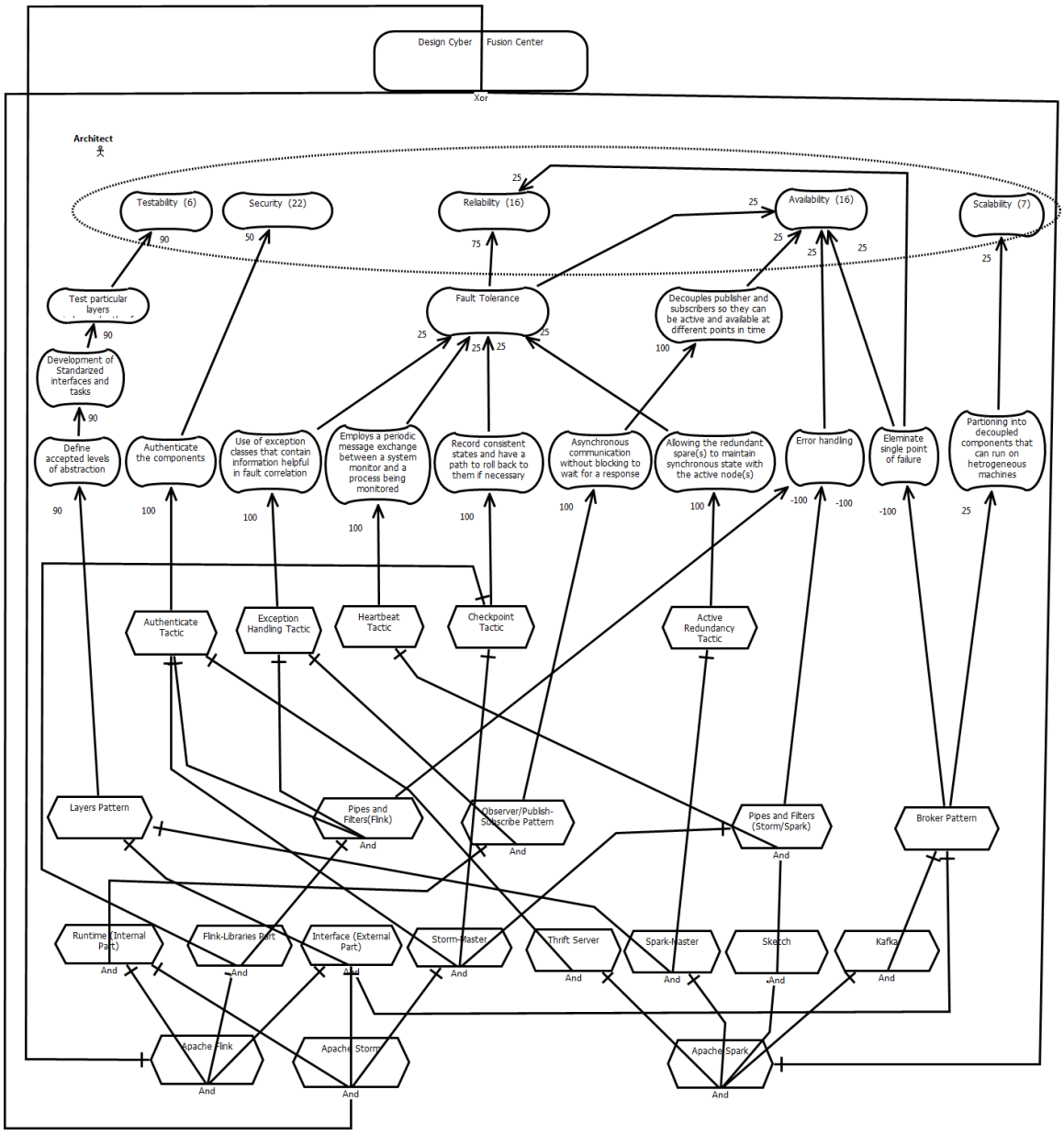


Figure 31 GRL model of Flink, Storm, and Spark in terms of theTestability, Security, Reliability, Availability, and Scalability requirements

The high-level goal of the project is shown at the top of the model connected to alternative candidate frameworks at the bottom of the model. Above each framework, framework parts are connected to their implemented patterns and tactics. The design decisions provide an explanation of why a pattern/tactic impacts an NFR the way it does at the top of the model. They contribute positively or negatively to the realization of NFRs. For example, applying

the ***Observer/Publish-Subscribe*** pattern pushes the Storm framework towards *Availability*. This is justified by the fact that the ***Observer/Publish-Subscribe*** pattern provides asynchronous communication between components without blocking to wait for a response, which helps decouple publishers and subscribers so that they can be active and available at different points in time, resulting in improving *Availability*. The implementation of the three reliability tactics: "*Checkpoint", "Exception Handling",* and *"Heartbeat"* also favor *Reliability* and *Availability* in Storm.

The ***Broker*** pattern in the Interface of Storm and Flink frameworks and the Kafka part of the Spark framework introduces a "*Single point of failure*" such that when the broker fails, the whole system stops working. This pulls the three frameworks away from *Reliability* and *Availability*. It is also hard to handle task/process errors by applying the ***Pipes and Filters*** pattern in the three frameworks because of the impossibility of, for example, restarting a pipeline or ignoring an error. This hurts *Availability* in the three frameworks. At the same time, applying the *"Heartbeat"* tactic in Spark and Storm results in improving the availability of these frameworks. The *"Checkpoint"* and *"Exception Handling"* tactics are only implemented in Storm and Flink frameworks. As I see also that all the tactics which belong to the *Reliability* requirement have positive contribution values to the reliability of type **(100)**.

In terms of the *Security*, the *"Authenticate"* tactic is implemented in all three frameworks. As calculated in section 6.2, *Reliability* and *Availability* have the same importance value, which is **(16)**, while *Testability* has the importance value **(6)**, *Security* has **(22)**, and *Scalability* has **(7)**.

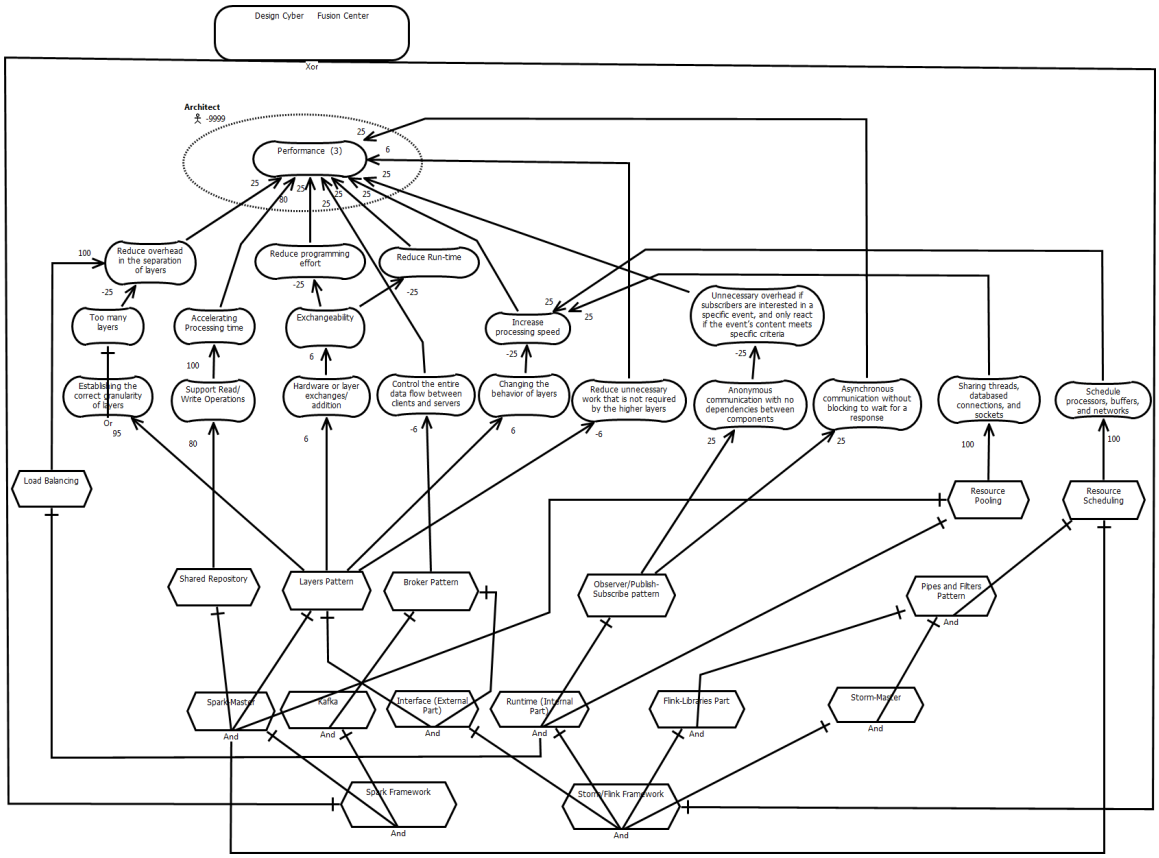


Figure 32 GRL model of Flink, Storm, and Spark in terms of Performance

Figure 32 shows the model for performance. The ***Layers*** pattern in all the three frameworks hurts the performance. Too many layers add overhead that increase the run-time, the programming effort, and decrease the processing speed. The ***Shared-Repository*** pattern promotes the performance by supporting read/write operations in a way that decreases the processing time. ***Shared-Repository*** therefore has a positive contribution of value **(80)** with the *Performance*. Tactics *"Resource Pooling"* and the *"Resource Scheduling"* improve the performance as well. They both have positive contribution values of type **(100)** with the *Performance*.

7.1.6. Evaluate the Software Architectures

In this step, I evaluate the software architectures models and derive a general evaluation of the architectures.

7.1.6.1. Evaluate the Models of the Software Architectures

I perform an evaluation of the models to calculate the satisfaction levels of the NFRs (Figures 31 and 32). The evaluation is done by applying different evaluation strategies on the GRL models. Figures 33, 34, and 35 show three different strategies before running the jUCMNav propagation and getting the satisfaction levels. The first strategy in Figure 33 models the application of Apache Spark where only its patterns and tactics are initially satisfied. Figure 34 models the application of Apache Storm, while Figure 35 models the application of Apache Flink.

Figures 36, 37, and 38 show the evaluation strategies, for Spark, Storm, and Flink respectively, after running the jUCMNav propagation and getting the satisfaction levels for each node in the model.

For the performance, Figures 39 and 40 show two different strategies before running the jUCMNav propagation and getting the satisfaction levels. Figure 39 models the application of both Storm and Flink. Note that, Storm and Flink implement the same patterns and tactics. This is why I merge their models in one model in terms of performance. Figure 40 models the application of Spark where only its patterns and tactics are initially satisfied.

Figure 41 shows the evaluation strategy for both Storm and Flink, after running the jUCMNav propagation and getting the satisfaction levels for each node in the model while Figure 42 shows the evaluation strategy for Spark.

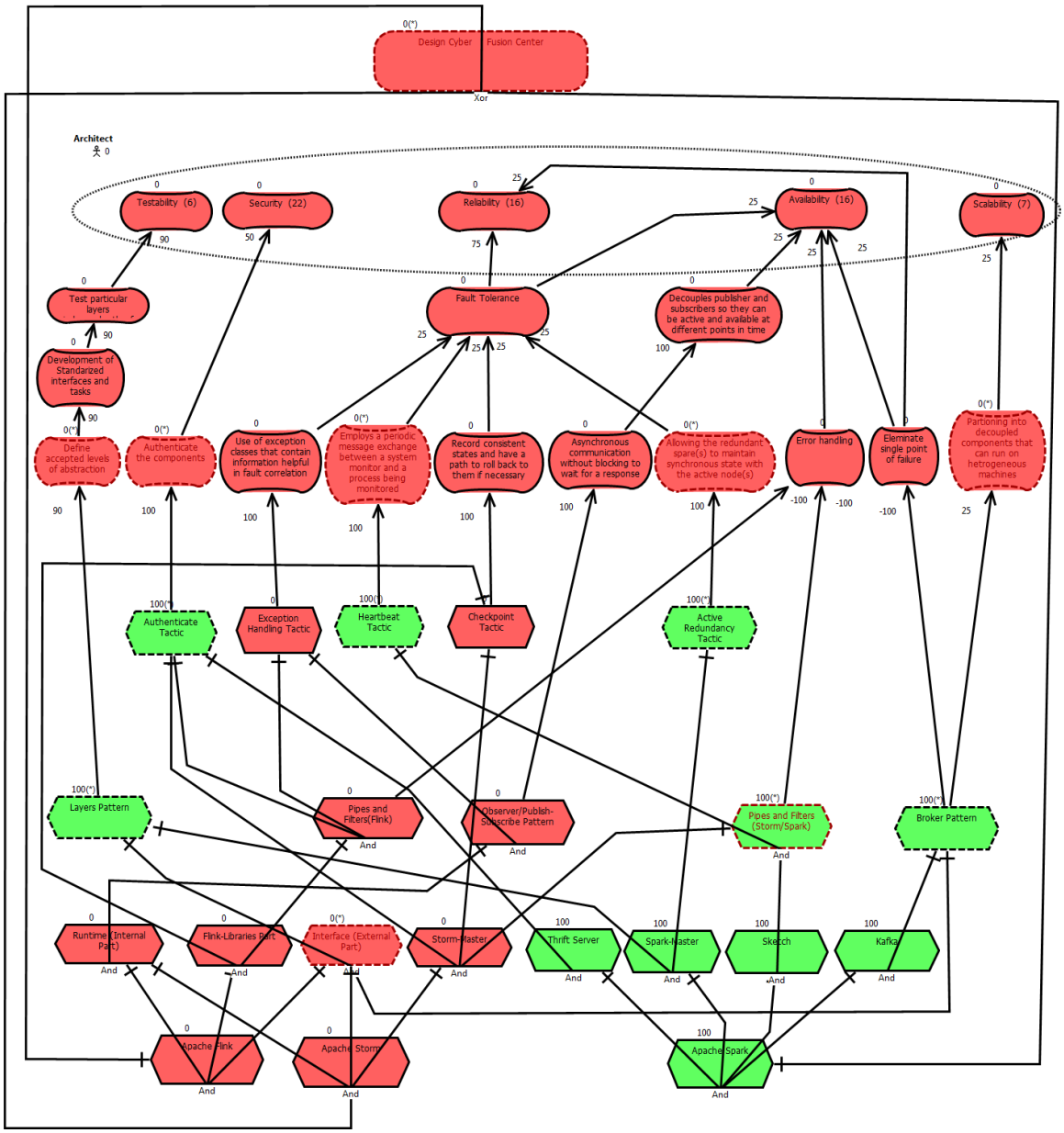


Figure 33 GRL model where patterns/tactics of Spark are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability before running the jUCMNav propagation

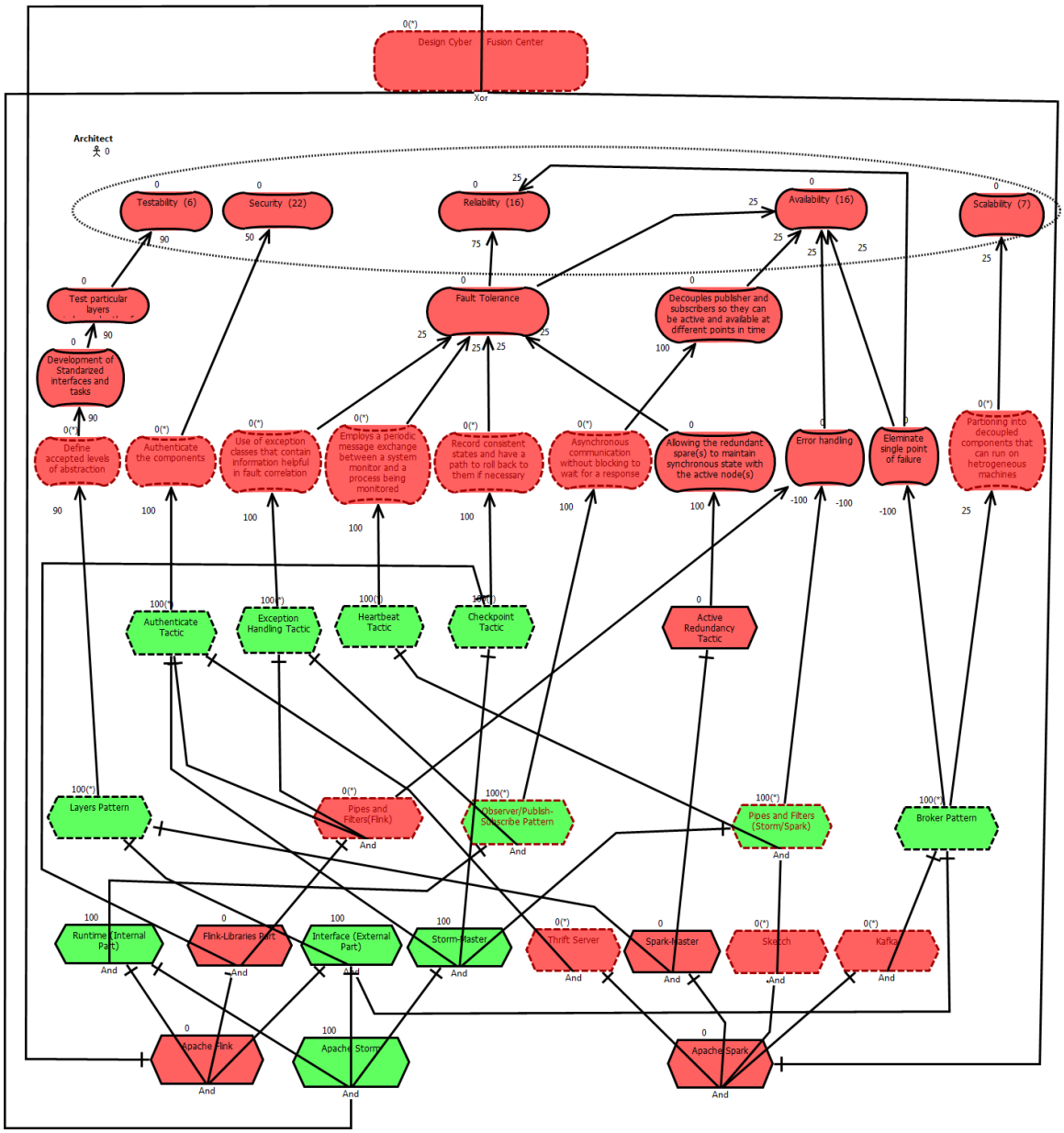


Figure 34 GRL model where patterns/tactics of Storm are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability before running the jUCMNav propagation

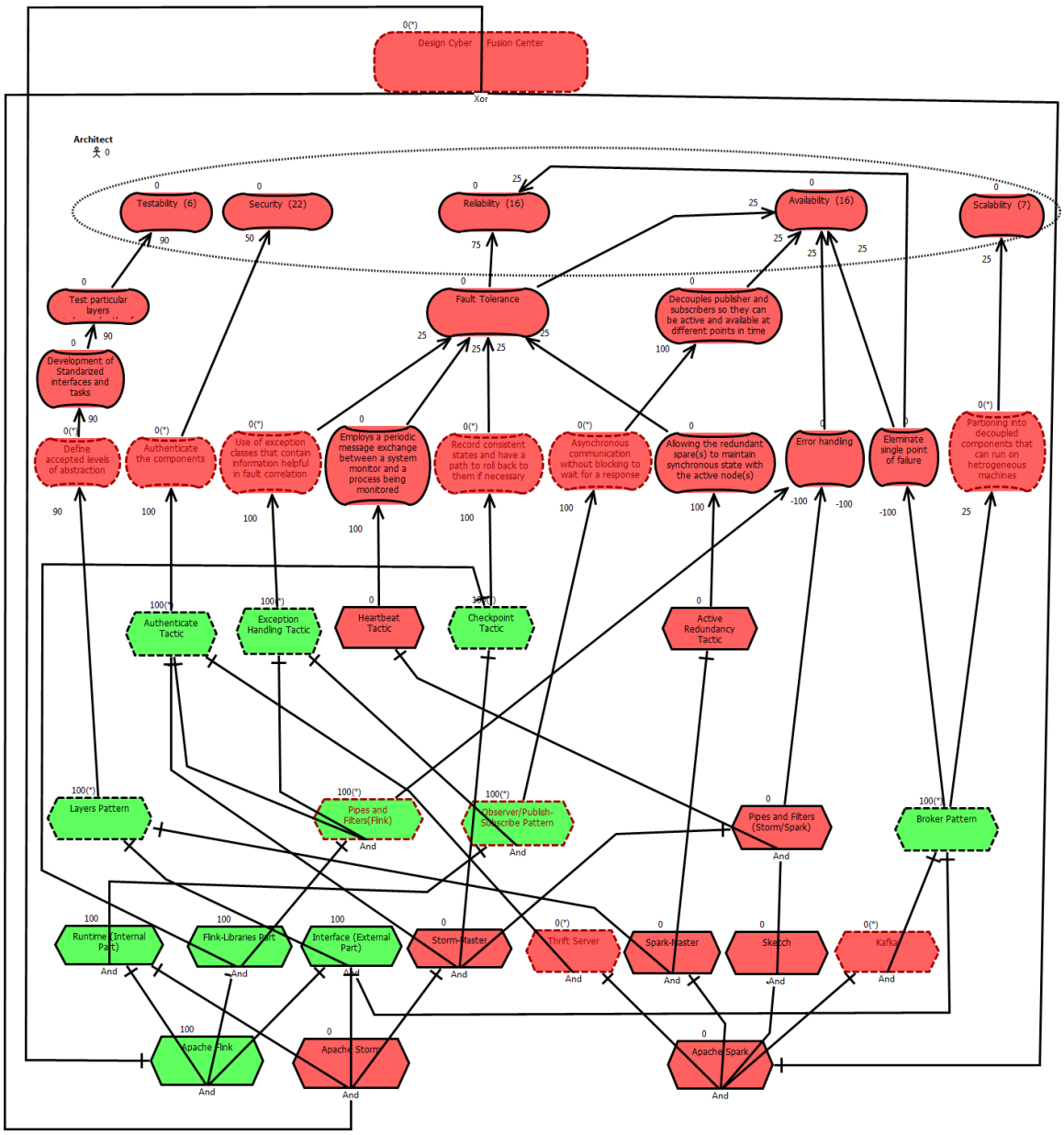


Figure 35 GRL model where patterns/tactics of Flink are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability before running the jUCMNav propagation

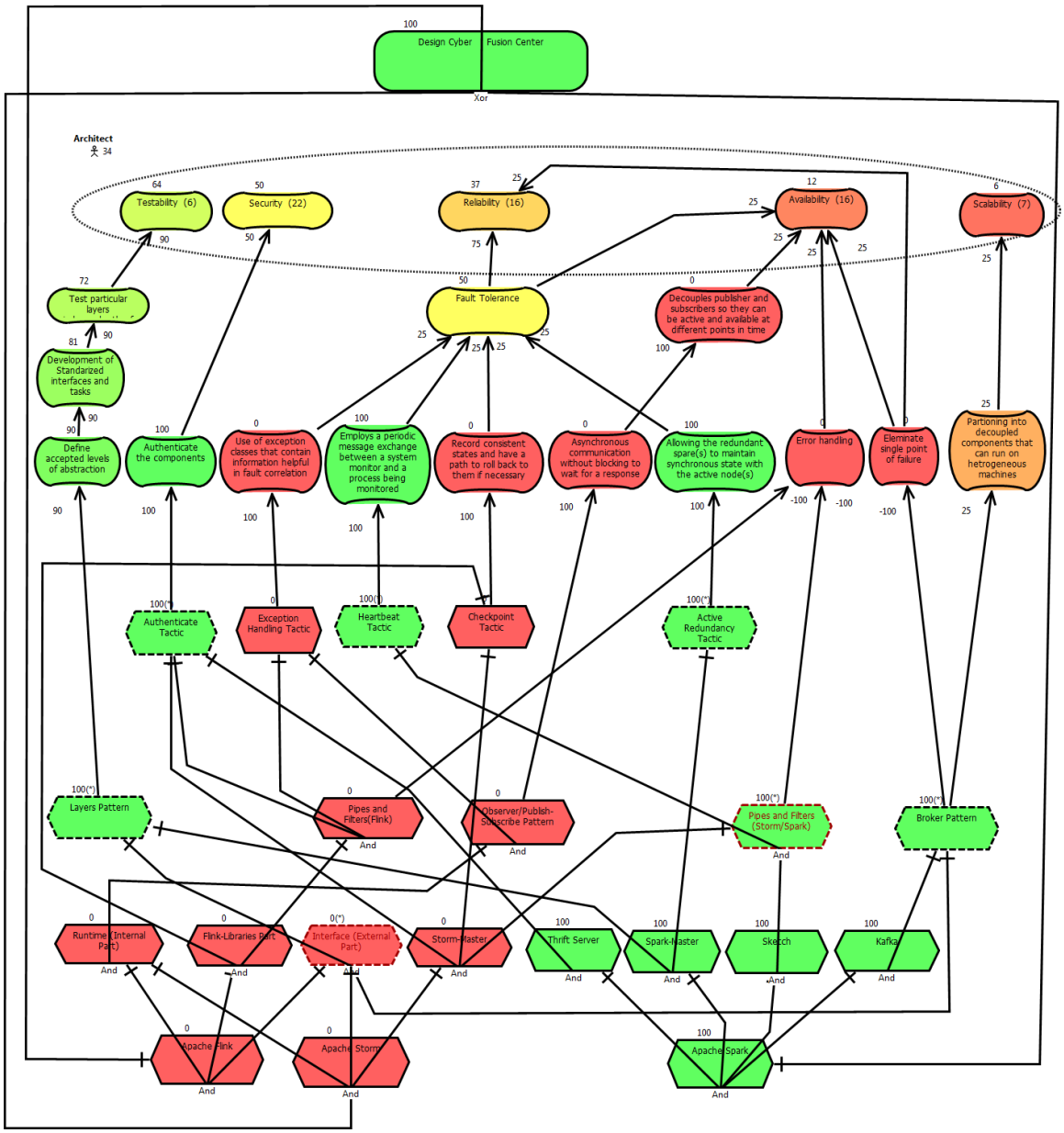


Figure 36 Strategy 1: Evaluated GRL model where patterns/tactics of Spark are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability after running the jUCMNav propagation

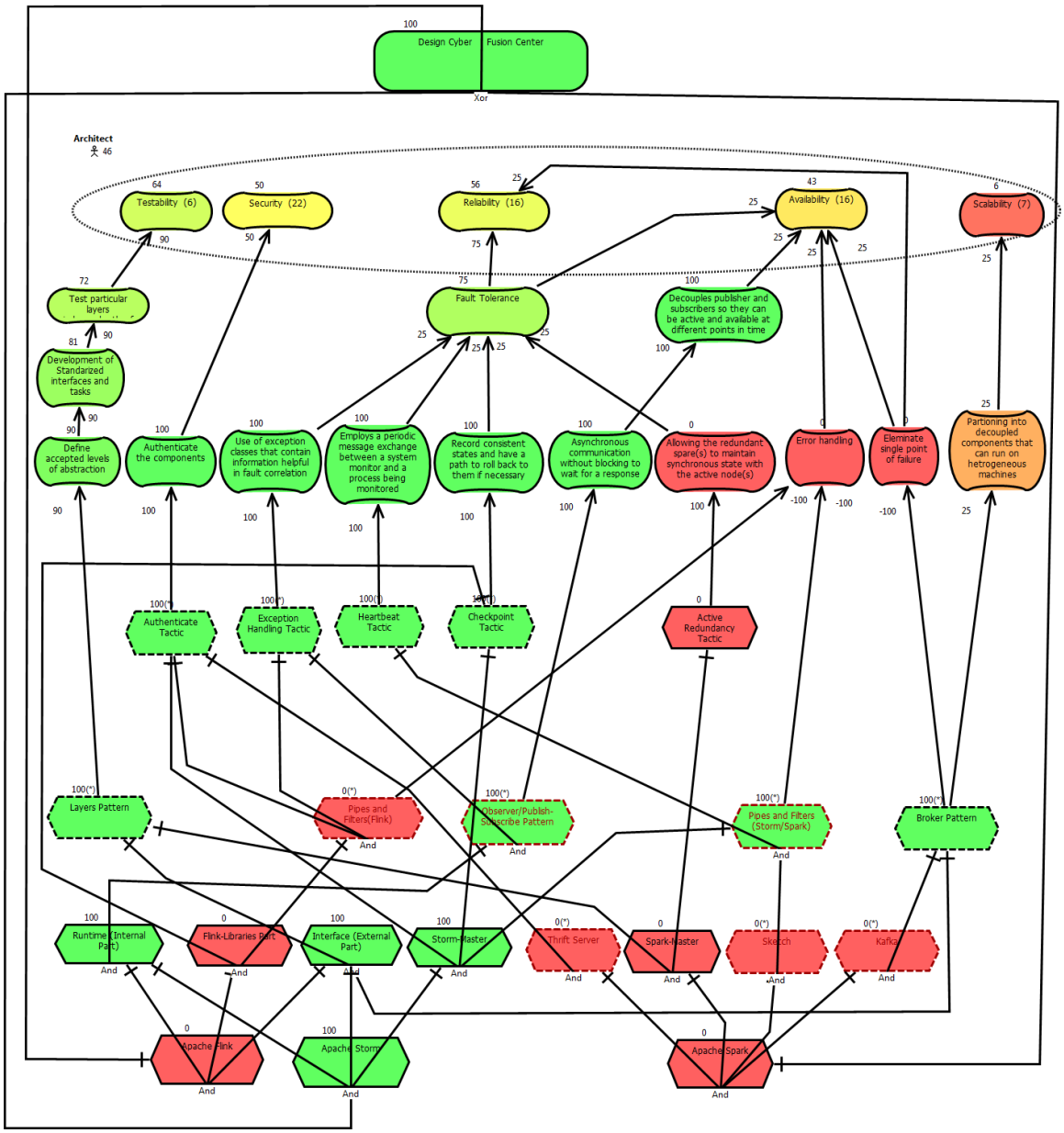


Figure 37 Strategy 2: Evaluated GRL model where patterns/tactics of Storm are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability after running the jUCMNav propagation

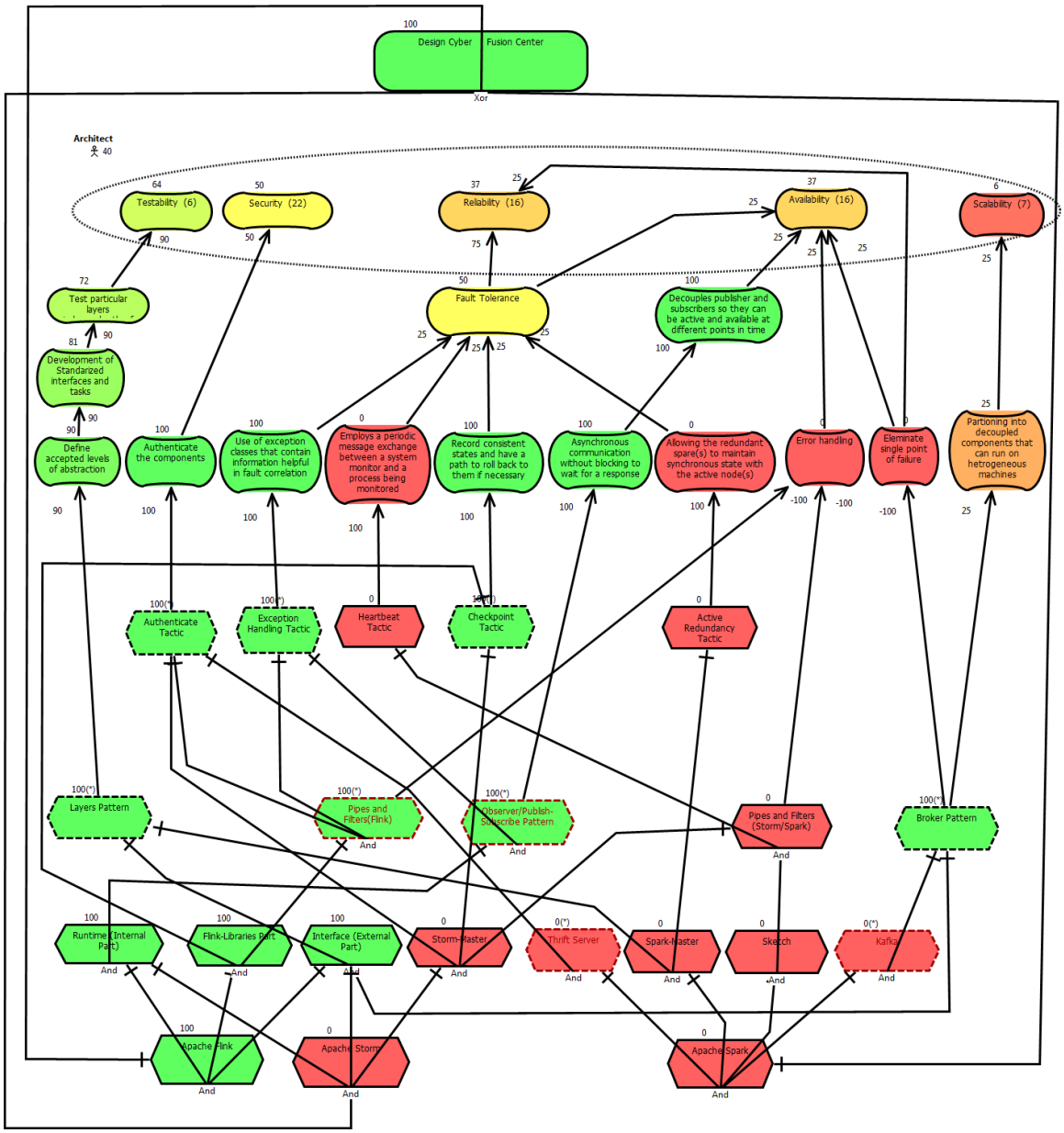


Figure 38 Strategy 3: Evaluated GRL model where patterns/tactics of Flink are initially satisfied in terms of Testability, Security, Reliability, Availability, and Scalability after running the jUCMNav propagation

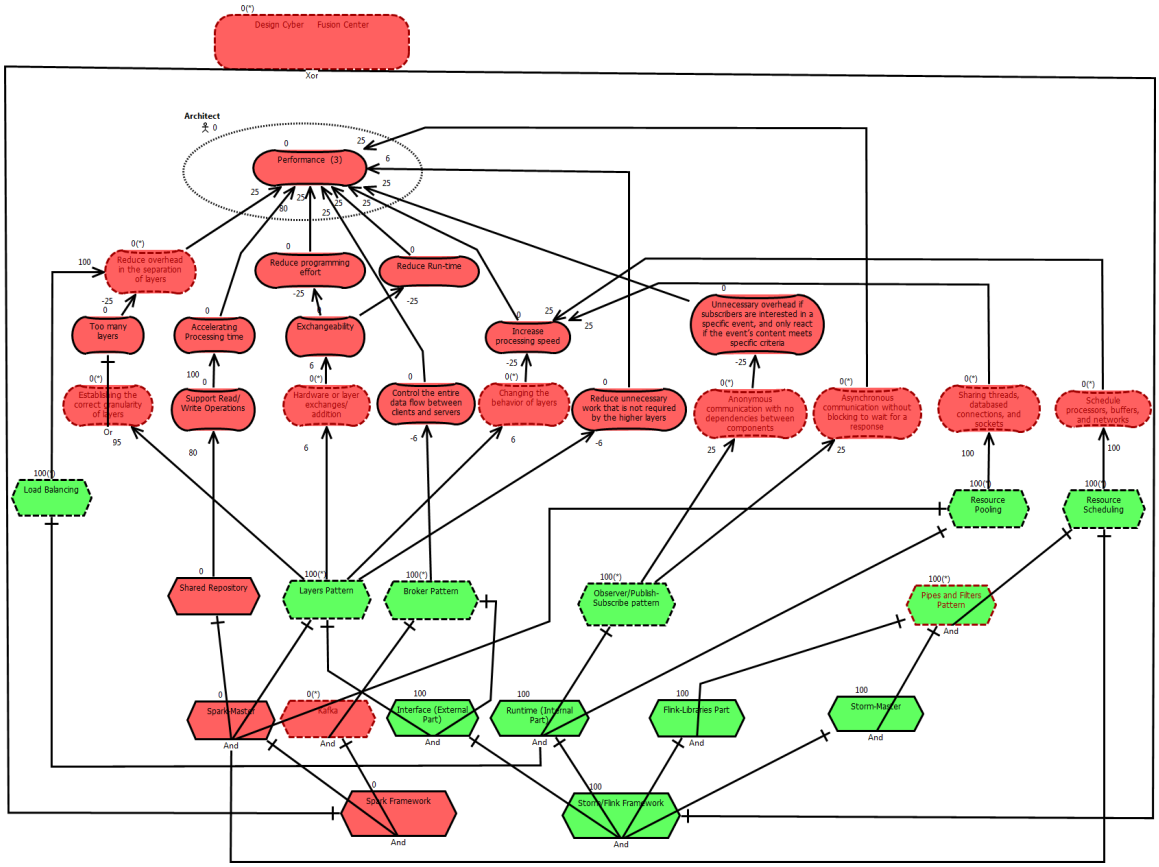


Figure 39 GRL model where patterns/tactics of Storm and Flink are initially satisfied in terms of Performance before running the jUCMNav propagation

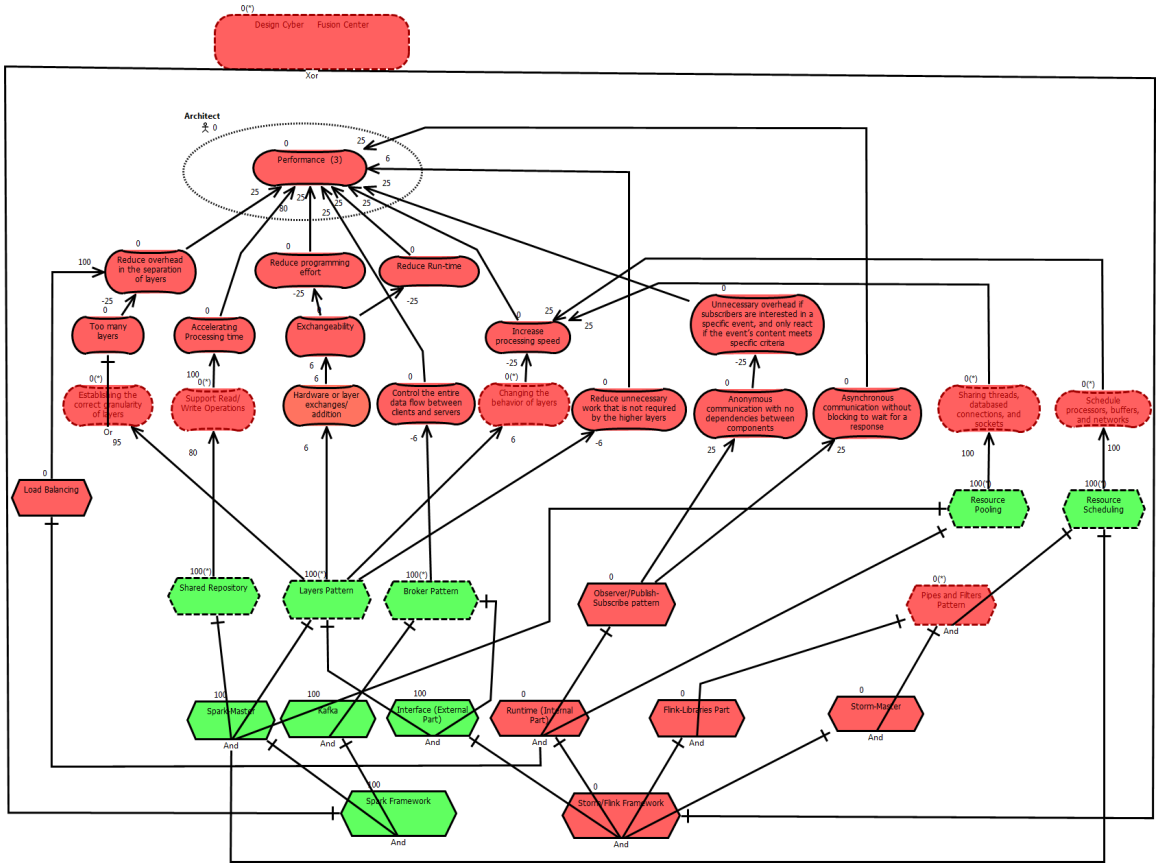


Figure 40 GRL model where patterns/tactics of Spark are initially satisfied in terms of Performance before running the jUCMNav propagation

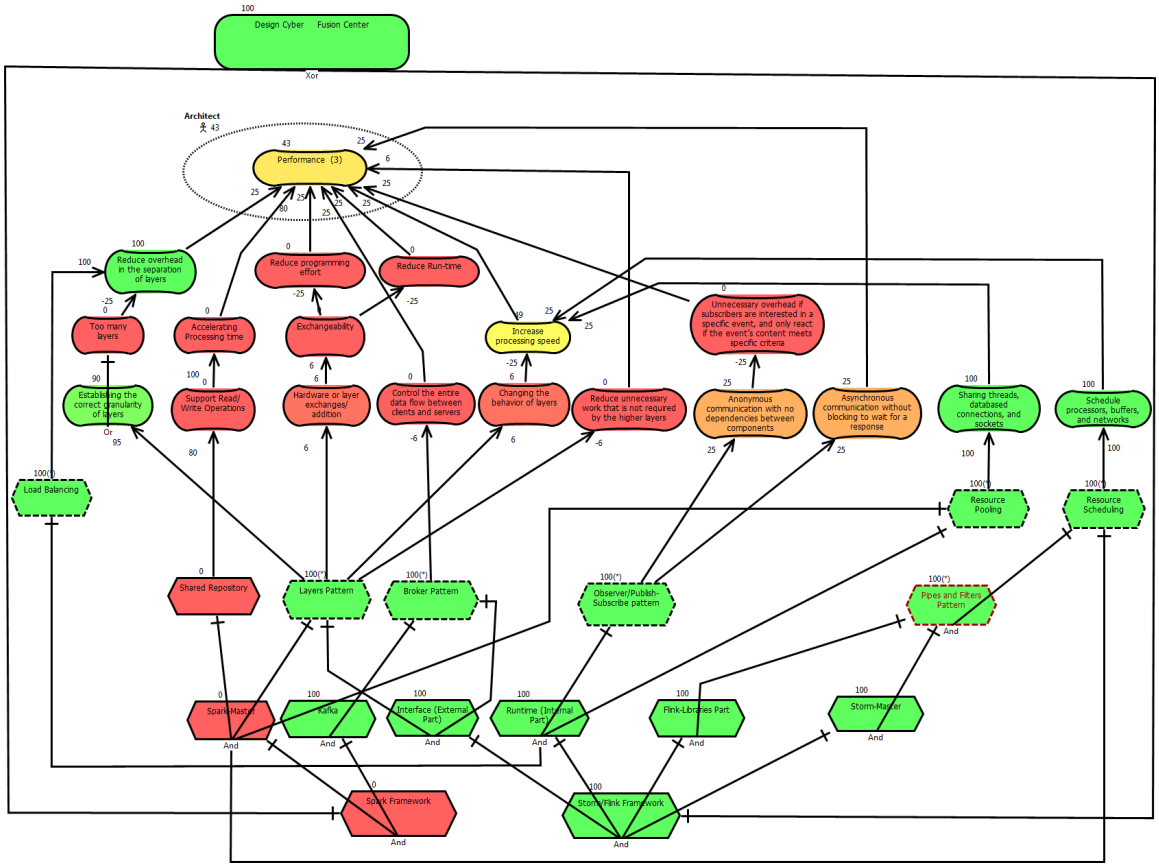


Figure 41 Strategy 1: Evaluated GRL model where patterns/tactics of Storm and Flink are initially satisfied in terms of Performance after running the jUCMNav propagation

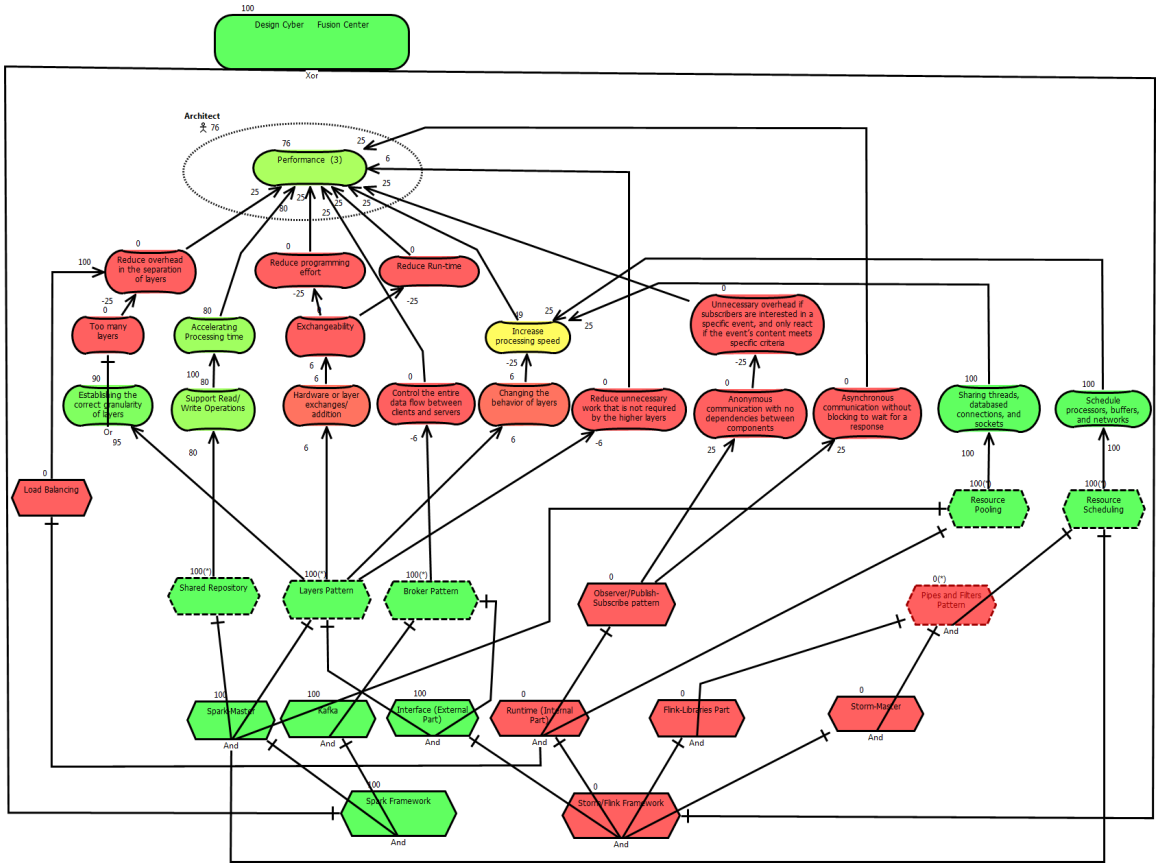


Figure 42 Strategy 2: Evaluated GRL model where patterns/tactics of Spark are initially satisfied in terms of Performance after running the jUCMNav propagation

7.1.6.2. Evaluate the Software Architectures

Based on the evaluation results of the GRL models in Figures 36, 37, and 38, I can notice that the three frameworks have similar satisfaction levels for the *Testability, Security, and Scalability* requirements. The *Testability* requirement is satisfied with **(64)** satisfaction level for all the frameworks, while the *Security* is satisfied with **(50)** satisfaction level and *Scalability* with **(6)** satisfaction level for all the frameworks. The Storm framework has higher satisfaction levels for *Reliability* and *Availability*, which are **(56)** and **(43)**, respectively, compared to Spark and Flink. This is because of the implementation of the three reliability tactics: "*Exception Handling", "Heartbeat",* and *"Checkpoint"*. They all improve fault tolerance, which improves reliability. Spark and Flink have the same satisfaction level for *Reliability*, which is **(37)**. Spark has the least satisfaction level for *Availability,* is **(12)**, while Storm has **(43)** and Flink has **(37)** satisfaction levels for

Availability. This is because of the application of the ***Observer/Publish-Subscribe*** pattern in Storm and Flink. It provides asynchronous communication between components. This helps decouple publishers and subscribers so that they can be active and available at different points in time, resulting in improving the availability of the frameworks. Both Storm and Flink use the *"Checkpoint"* tactic to Record consistent states and have a path to roll back to them if necessary, while Spark uses the "*Active Redundancy"* tactic for recovery, preparation, and repair of the errors.

The architect is more satisfied with Storm than Flink and Spark. The satisfaction value for Storm is **(46)**, while it is **(40)** for Flink and **(34)** for Spark. If an architect favors Reliability and Availability over the other requirements, our approach would recommend Storm. However, if *Testability, Security,* and *Scalability* are preferred, then any one of the three frameworks could be equally recommended.

In terms of the performance and based on the evaluation results of the GRL models in Figures 41 and 42, Spark has better performance compared to Storm and Flink. Spark achieves optimal processing time by using the memory (the ***Shared-Repository*** pattern) to store the intermediate results as objects spread across a Spark cluster. While Storm and Flink send their intermediate results directly to the network through channels between the workers. This makes the processing time very dependent on the cluster's local network. The read/write operations with the ***Shared Repository*** pattern also help to accelerate the processing time. The satisfaction level of the performance for Spark is **(76)** while it is **(43)** for both Storm and Flink. Strom and Flink are therefore, quite similar in terms of the data processing speed, as shown in Figures 41 and 42. Applying the *"Resource Pooling"* and the *"Resource Scheduling"* tactics also improves the performance in the three frameworks. The "Resource Pooling" tactic helps to share threads, database connections, and sockets, and the *"Resource Scheduling"* tactic helps schedule processors, buffers, and networks. Applying the ***Pipes and Filters***, which implements the *"Resource Scheduling"* helps to minimize the period of idle resources, which maximize the use of Central Processing Unit (CPU) resources in both Storm and Flink frameworks. If an architect favors *Performance* over other requirements, I would recommend the Spark framework.

7.2. Case Study 2: Healthcare Supportive Home-System of Systems (HSH-SoS)

Healthcare Supportive Home-System of Systems (HSH-SoS) [106] are collaborative System of Systems (SoS), where each constituent system maintains its operational and managerial independence, but collaborate with a Healthcare Supportive Home system (HSH) to achieve and evolve the global missions. Examples of constituent systems include smart homes, telemedicine systems, teleconsultation systems, emergency systems, activity monitoring systems, physiological monitoring systems, smart devices, company robots, mobile apps, smart TV apps, and rehabilitation systems, among others [106]. HSH-SoS provide health services to patients in their residence and supply them with autonomy through the interaction and coordination of their constituent systems.

The first case study (cyber fusion center) discusses the applicability of our approach for evaluating different architectures (i.e. frameworks in the case study) in order to compare and select the best suited alternative. This case study (HSH-SoS) discusses another application, which is, the evaluation of an architecture and formulation of a rationale in terms of its implemented patterns and tactics and their impacted NFRs. This type of evaluation may be used to analyze and understand the implementation of an architecture. It may provide architects an understanding of the design of the software architecture in terms of quality attributes (i.e. if they will achieve deadlines in real time systems and the kind of modifications or changes supported).

Note that, I modeled the six determined patterns in the context of SoS (shown in Table 17 in Chapter 4) and added them to the GRL inventory. The models of those patterns are shown in Appendix B.

7.2.1. Determining the Patterns and Tactics Implemented in A Software Architecture

In this step, I apply *Archie* to the HSH-SoS architecture to determine its patterns and tactics, as shown in the following sections.

7.2.1.1. Patterns and Tactics Detected with HSH-SoS

I applied *Archie* on the HSH-SoS architecture and got a set of candidate patterns and tactics, as shown in Table 35.

Table 35 Analysis of applying Archie to the HSH-SoS architecture

#	Tactics	Number of Trained in Keywords	Number of Detected Keywords	Frequency of the Detected Terms	Frequency of Tactic	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	0	0	0	NO	NA	-
2	**Heartbeat**	13	0	0	0	NO	NA	-
3	**Ping/Echo**	10	1	1	0	Yes	TP	<=0.2
4	**Exception Handling**	5	3	3	2	Yes	TP	<=0.3
5	**Authenticate**	10	0	0	0	NO	NA	-
6	**Time Stamp**	4	2	153	10	Yes	TP	<=0.7
7	**Resource Pooling**	10	2	1	1	Yes	TP	<=0.6
8	**Audit Trail**	10	0	0	0	NO	NA	-
9	**PBAC**	10	1	7	0	Yes	FP	<=0.3
10	**RBAC**	10	1	4	0	Yes	FP	<=0.3
11	**Resource Scheduling**	10	1	3	1	Yes	TP	<=0.3
12	**Session Management**	10	1	2	0	Yes	FP	<=0.2
13	**Load Balancing**	10	0	0	0	NO	NA	-
14	**Restart**	4	2	15	2	Yes	TP	<=0.8
15	**Time-Out**	3	2	151	16	Yes	TP	<=0.2
16	**Cancel**	1	1	9	2	Yes	TP	<=1
17	**Active Redundancy**	10	2	37	2	Yes	TP	<=0.4
18	**Checkpoint**	12	0	0	0	NO	NA	-
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained in Keywords**	**Number of Detected Keywords**	**Frequency of the Detected Terms**	**Frequency of Pattern**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	32	5	Yes	TP	<=0.8
2	**Broker**	10	0	0	0	NO	NA	-
3	**Observer/Publish-Subscribe**	4	2	11	4	Yes	TP	<=0.2
4	**Pipes and Filters**	3	1	8	4	Yes	TP	<=0.4
5	**Shared-Repository**	4	0	0	0	NO	NA	-
6	**Centralized Architecture**	4	1	6	0	Yes	FP	<=0.2
7	**Service-Oriented Architecture**	7	1	33	10	Yes	TP	<=0.2
8	**Enterprise Service Bus**	7	1	15	2	Yes	TP	<=0.2
9	**Trickle-Up Software pattern**	5	3	46	6	Yes	TP	<=0.4
10	**Reconfiguration Control Architecture**	5	0	0	0	NO	NA	-
11	**Contract Monitor**	3	1	13	2	Yes	TP	<=0.3
12	**Pace-Layers**	6	0	0	0	NO	NA	-

13	**Reflective Architecture**	5	4	66	12	Yes	TP	<=0.5

Figure 43 shows a sample of the detected patterns and tactics when I run the *Archie* tool on the HSH-SoS source code.

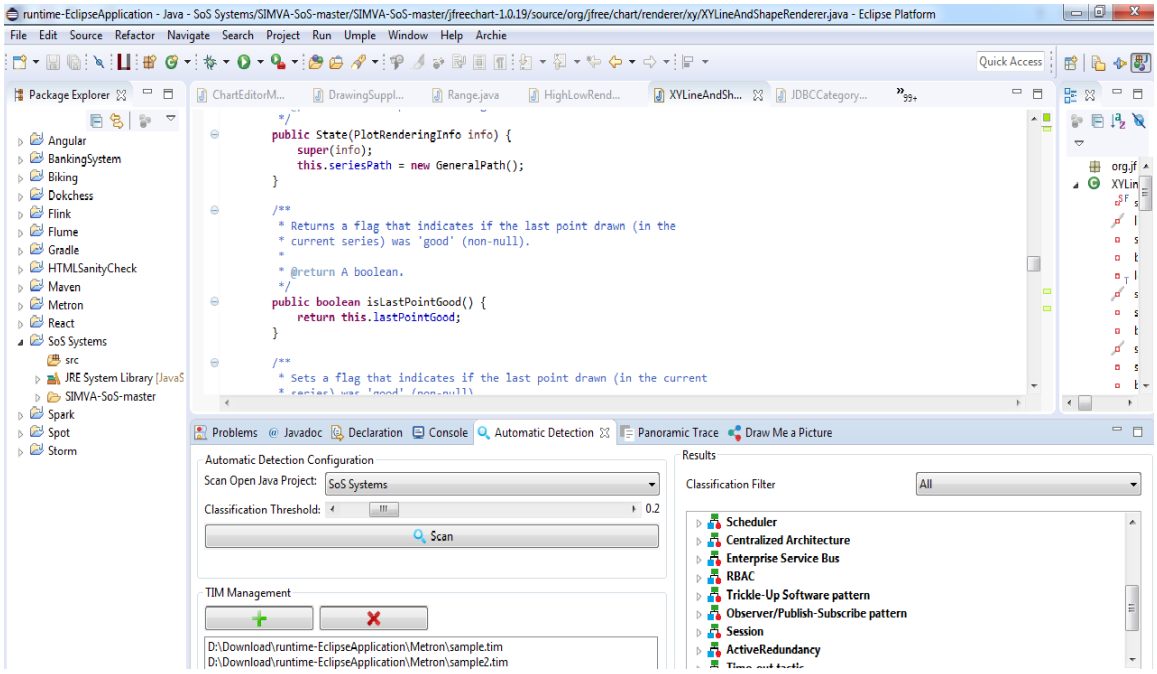
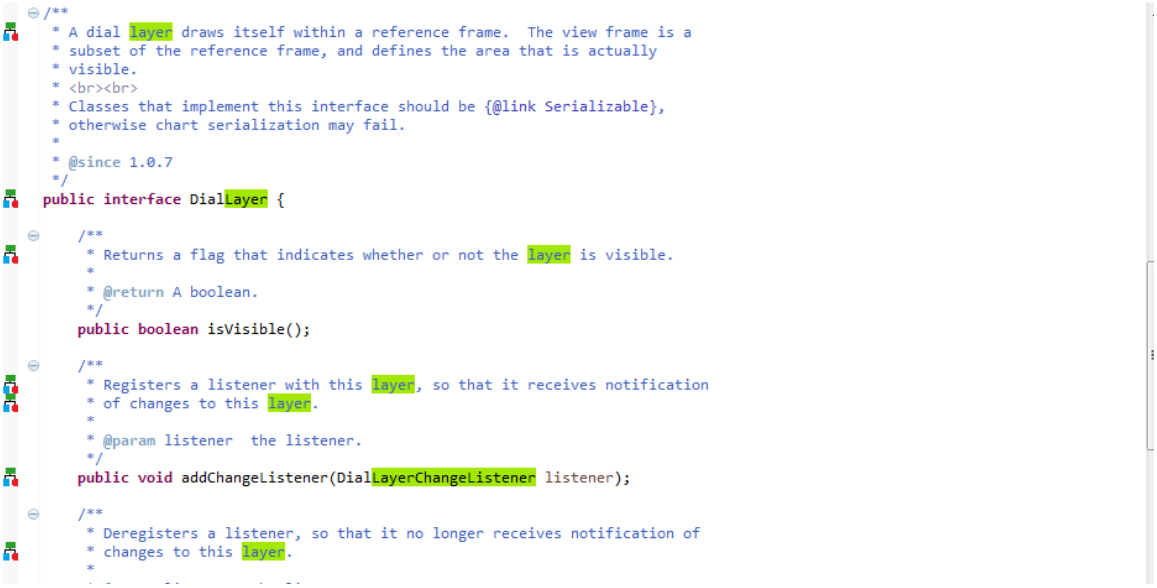


Figure 43 Sample of the detected patterns/tactics for HSH-SoS Architecture

I then validated the results of HSH-SoS architecture by looking manually for occurrences of the detected patterns/tactics in the source code/documentation/websites of the HSH-SoS architecture. For example, Figure 44 shows the source code where the ***Layers*** pattern is detected in HSH-SoS implementation and its related terms.

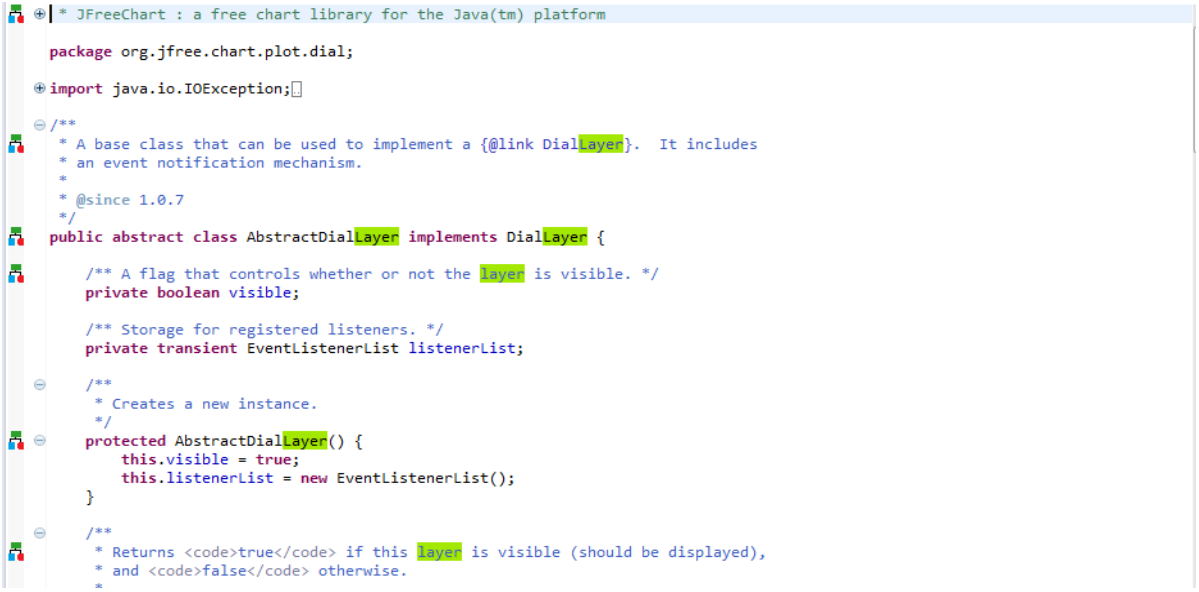


```java
/**
 * A dial layer draws itself within a reference frame.  The view frame is a
 * subset of the reference frame, and defines the area that is actually
 * visible.
 * <br><br>
 * Classes that implement this interface should be {@link Serializable},
 * otherwise chart serialization may fail.
 *
 * @since 1.0.7
 */
public interface DialLayer {

    /**
     * Returns a flag that indicates whether or not the layer is visible.
     *
     * @return A boolean.
     */
    public boolean isVisible();

    /**
     * Registers a listener with this layer, so that it receives notification
     * of changes to this layer.
     *
     * @param listener  the listener.
     */
    public void addChangeListener(DialLayerChangeListener listener);

    /**
     * Deregisters a listener, so that it no longer receives notification of
     * changes to this layer.
     *
```



```java
* JFreeChart : a free chart library for the Java(tm) platform

package org.jfree.chart.plot.dial;

import java.io.IOException;

/**
 * A base class that can be used to implement a {@link DialLayer}.  It includes
 * an event notification mechanism.
 *
 * @since 1.0.7
 */
public abstract class AbstractDialLayer implements DialLayer {

    /** A flag that controls whether or not the layer is visible. */
    private boolean visible;

    /** Storage for registered listeners. */
    private transient EventListenerList listenerList;

    /**
     * Creates a new instance.
     */
    protected AbstractDialLayer() {
        this.visible = true;
        this.listenerList = new EventListenerList();
    }

    /**
     * Returns <code>true</code> if this layer is visible (should be displayed),
     * and <code>false</code> otherwise.
     *
```

Figure 44 Source code where Layers pattern is detected in HSH-SoS and its related terms

The numbers of the True Positives (TP), False Positives (FP), and accordingly the precision for HSH-SoS are shown in Table 36 (Table 20 in Chapter 4).

Table 36 Numbers of TP, FP, and Precision for HSH-SoS

System	TP	FP	Precision
HSH-SoS	16	4	0.80

7.2.1.2. Overlaps between Patterns and Tactics

In this step, I determine the overlaps between patterns and tactics of the HSH-SoS architecture, as shown in Table 37.

Table 37 Results of the overlaps in the HSH-SoS architecture

Pattern	Shared Java Class(s)	Tactic(s)	Overlaps
Service-Oriented Architecture (SOA)	Axis.java PrismSyntaxHighlighter.java SoSSlicer.java	Exception Handling	3
	SoSSlicer.java	Close	1
Centralized Architecture	-	-	-

Enterprise Service Bus	SegmentedTimelineTest.java	Timeout, Exception Handling	1
Trickle-Up Software pattern	-	-	-
Reconfiguration Control Architecture	-	-	-
Contract Monitor	Marker.java	Authentication	1
	Organization.java	Authorization	1
	Timeline.java	Timeout	1
Pace-Layers	-	-	-
Reflective Architecture	SunJPEGEncoderAdapter.java	Close, Exception Handling	1
	TimeSeries.java RegularTimePeriod.java	Timeout	2
Pipes and Filters	-	-	0
Observer/Publish-Subscribe	-	-	0

As I can in Table 37, the *"Exception Handling"* tactic has three classes and the Close tactic has a single class overlapped with the ***SOA*** pattern. Each of the *"Authentication",* the *"Authorization",* and the *"Timeout"* has a single class overlapped with the ***Contract Monitor*** pattern. The ***Centralized Architecture***, ***Trickle-Up Software***, ***Reconfiguration Control Architecture, Pace-Layers, Pipes and Filters,*** and ***Observer/Publish-Subscribe*** patterns have no overlaps with any tactic in the HSH-SoS architecture.

7.2.2. Modeling Software Architectures in terms of their Implemented Patterns and Tactics

I modelled the HSH-SoS architecture in terms of its implemented patterns and tactics by performing the following sub-steps:

a. Identify the most architecturally significant NFRs to be considered in the design of software architectures in the determined context.

b. Calculate the importance values of the considered NFRs.

c. Model software architectures in terms of their implemented patterns and tactics.

7.2.2.1. Identify the Most Architecturally Significant NFRs to be Considered in the Design of Software Architectures in the Determined Context

A set of the commonly used quality attributes for SoSs are reported in [106]. I extended this list by searching about the NFRs or quality attributes that are addressed in SoSs. Our results are shown in Table 38 and Table 62 in Appendix A. I filtered those NFRs to the HSH-SoS architecture's NFRs, as shown in Table 39.

Table 38 Relative NFRs for SoS

Non-Functional Requirements (NFRs)	
Authorization	Usability
Adaptability	Maintainability
Interoperability	Security
Reliability	Portability
Scalability	Usability
Deployability	Efficiency
Adaptivity	Performance
Configurability	Compatibility

Table 39 Summary of the relative NFRs for HSH-SoS

Non-Functional Requirements (NFRs)
Interoperability
Security
Reusability
Maintainability
Efficiency
Performance
Scalability

7.2.2.2. Calculate the Importance Values of the NFRs

I followed the steps discussed in Section 6.2 to calculate the importance values of the considered NFRs of the HSH-SoS architecture.

Assuming a comparison by an architect of the importance level of only the five NFRs *Interoperability, Security, Reusability,* and *Maintainability* as follow:

Interoperability << < = > >> Security

Interoperability << < = > >> Reusability

Interoperability	<<	<	=	>	>>	Maintainability
Security	<<	<	=	>	>>	Reusability
Security	<<	<	=	>	>>	Maintainability
Reusability	<<	<	=	>	>>	Maintainability

Indicating that *Interoperability* is much more important than *Security, Reusability,* and *Maintainability*. *Security* is important than *Reusability* and *Maintainability*. *Reusability* is less important than *Maintainability*.

The next step is to represent this in the matrix, as shown in Table 40, to show the pairwise comparison.

Table 40 Pairwise comparison matrix

	Interoperability	**Security**	**Reusability**	**Maintainability**
Interoperability	1	5	5	5
Security	1/5	1	3	3
Reusability	1/5	1/3	1	1/3
Maintainability	1/5	1/3	3	1

Then, the matrix is normalized.

- First, I add up all values in each column, as shown in Table 41.

Table 41 Addition of each column

	Interoperability	**Security**	**Reusability**	**Maintainability**
Interoperability	1	5	5	5
	+	+	+	+
Security	0.2	1	3	3
	+	+	+	+
Reusability	0.2	0.33	1	0.33
	+	+	+	+
Maintainability	0.2	0.33	3	1
Sum	**1.6**	**6.67**	**12**	**9.33**

- And then divide by corresponding column sum, as shown in Table 42.

Table 42 Divide each value by the sum of its column

	Interoperability	Security	Reusability	Maintainability
Interoperability	1/**1.6**	5/**6.67**	5/**12**	5/**9.33**
Security	0.2/**1.6**	1/**6.67**	3/**12**	3/**9.33**
Reusability	0.2/**1.6**	0.33/**6.67**	1/**12**	0.33/**9.33**
Maintainability	0.2/**1.6**	0.33/**6.67**	3/**12**	1/**9.33**

- I make sure the values in each column add up to 1, as shown in Table 43.

Table 43 New sum of each column

	Interoperability	Security	Reusability	Maintainability
Interoperability	0.625	0.75	0.42	0.53
Security	0.125	0.15	0.25	0.32
Reusability	0.125	0.05	0.08	0.04
Maintainability	0.125	0.05	0.25	0.11
Sum	**1**	**1**	**1**	**1**

Finally, I get the average of each row to get the corresponding rating (in this thesis, I use a ranking of 0 to 100 for GRL importance levels, so I multiply each row average by 100), as shown in Table 44.

Table 44 Calculated the importance level of each NFR relative to the other

	Interoperability	Security	Reusability	Maintainability	Row Average	X 100
Interoperability	0.625	0.75	0.42	0.53	0.58	**58**
Security	0.125	0.15	0.25	0.32	0.21	**21**
Reusability	0.125	0.05	0.08	0.04	0.07	**7**
Maintainability	0.125	0.05	0.25	0.11	0.14	**13**
Sum	**1**	**1**	**1**	**1**	**1**	**100**

Figure 45 below shows the calculated level of importance of the four NFRs.

Figure 45 Calculated importance levels of the NFRs

7.2.2.3. Model Software Architectures in terms of their Implemented Patterns and Tactics

I modeled the architecture of the HSH-SoS in terms of its implemented patterns and tactics considering the NFRs *Interoperability, Security, Reusability,* and *Maintainability*, as shown in Figure 46.

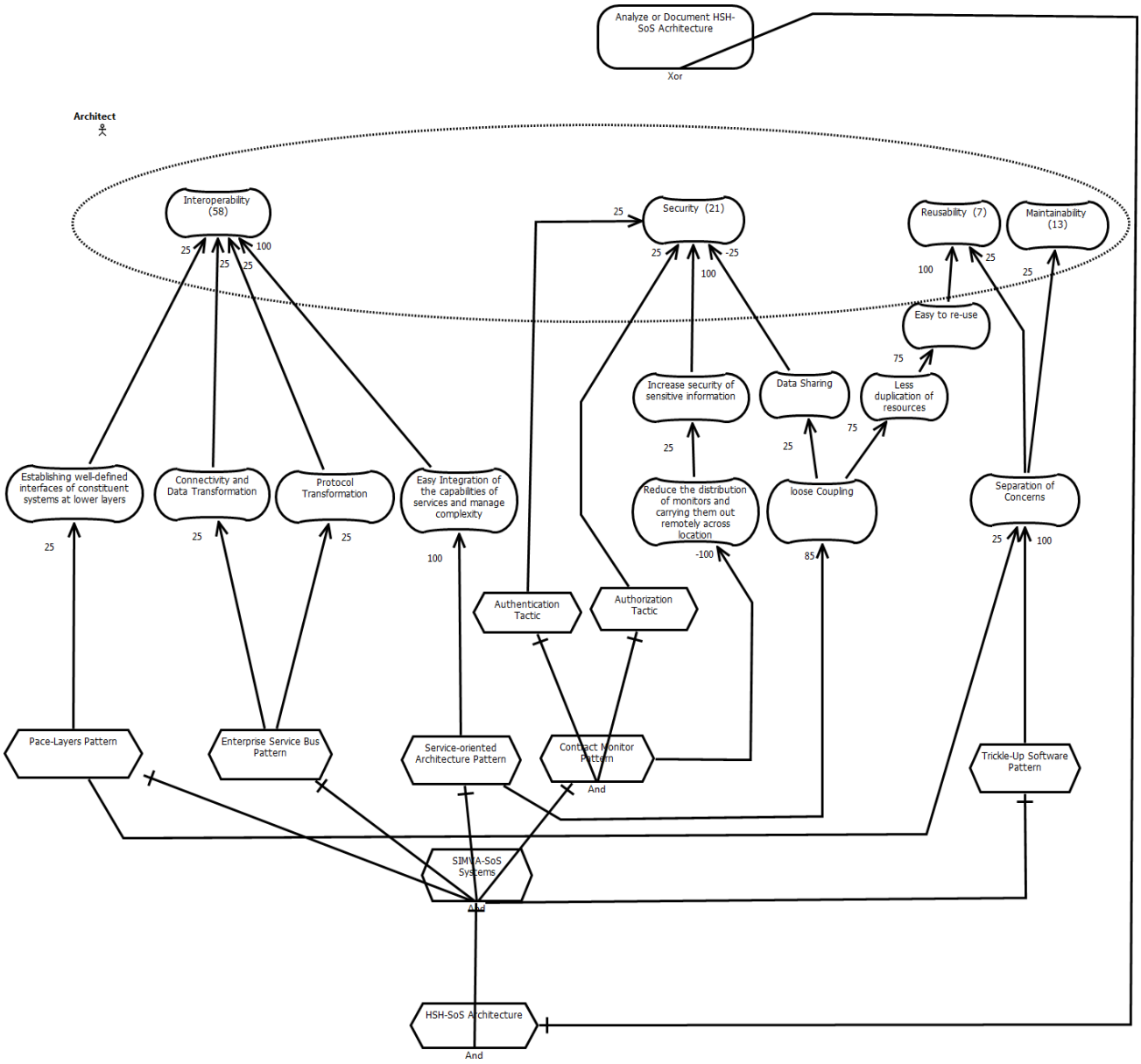


Figure 46 GRL model of the HSH-SoS architecture

The high-level goal is to analyze the architecture of HSH-SoS. The implemented patterns in the HSH-SoS architecture are the ***Pace-Layers,*** the ***Enterprise Service Bus,*** the ***Service-oriented Architecture,*** the ***Contract Monitor,*** and the ***Trickle-up Software***. The two security tactics, *"Authentication"* and *"Authorization"* are implemented in the ***Contract Monitor*** pattern in the HSH-SoS architecture. They push the system toward the security requirement. This is because all the security tactics have positive contributions of

type **(25)**. The ***Service-oriented Architecture*** and the ***Enterprise Service Bus*** patterns are the most important patterns in the HSH-SoS architecture. This is shown in the model where both patterns have several links with positive contributions to several NFRs. They push the HSH-SoS toward most of the considered requirements. Based on our calculations for the importance values of the NFRs of the HSH-SoS architecture in the previous step, the *Interoperability* has **(58)** importance value, while the *Security* has **(21)** importance value. The importance values of the *Reusability* and *the Maintainability* are **(7)** and **(13)**, respectively.

7.2.3. Evaluate Software Architectures

This step includes the following sub-steps:

1) Evaluate the model of the software architectures.
2) Evaluate the software architectures.

7.2.3.1. Evaluate the Model of the Software Architectures

I perform an evaluation of the model to calculate the satisfaction levels of the NFRs (Figure 46). The evaluation is done by an evaluation strategy on the GRL model. Figure 47 shows the strategy before running the jUCMNav propagation and getting the satisfaction levels for the HSH-SoS architecture.

Figure 48 shows the evaluation strategy, for HSH-SoS, after running the jUCMNav propagation and getting the satisfaction levels for each node in the model.

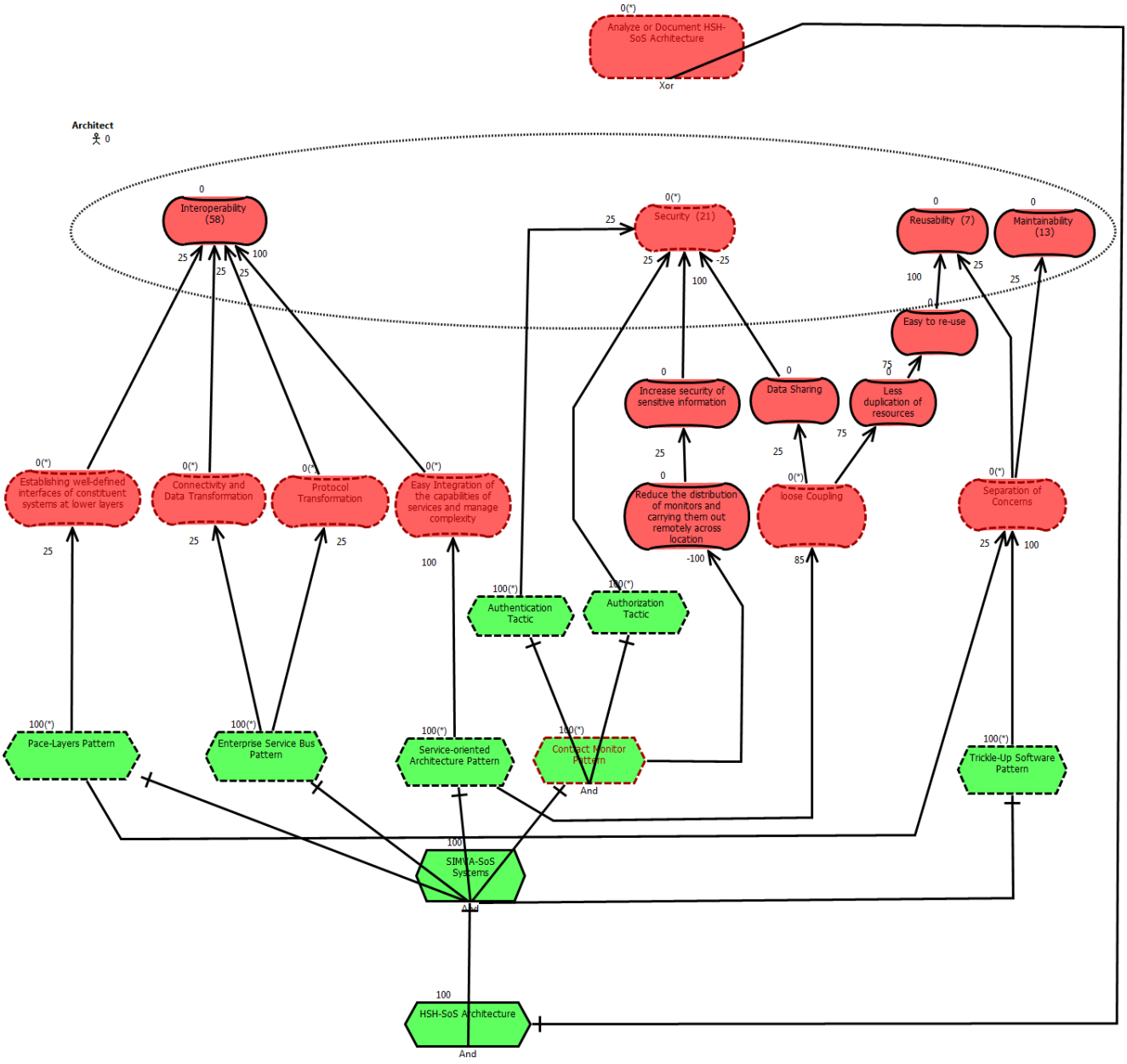


Figure 47 GRL model where patterns/tactics of HSH-SoS are initially satisfied before running the jUCMNav propagation

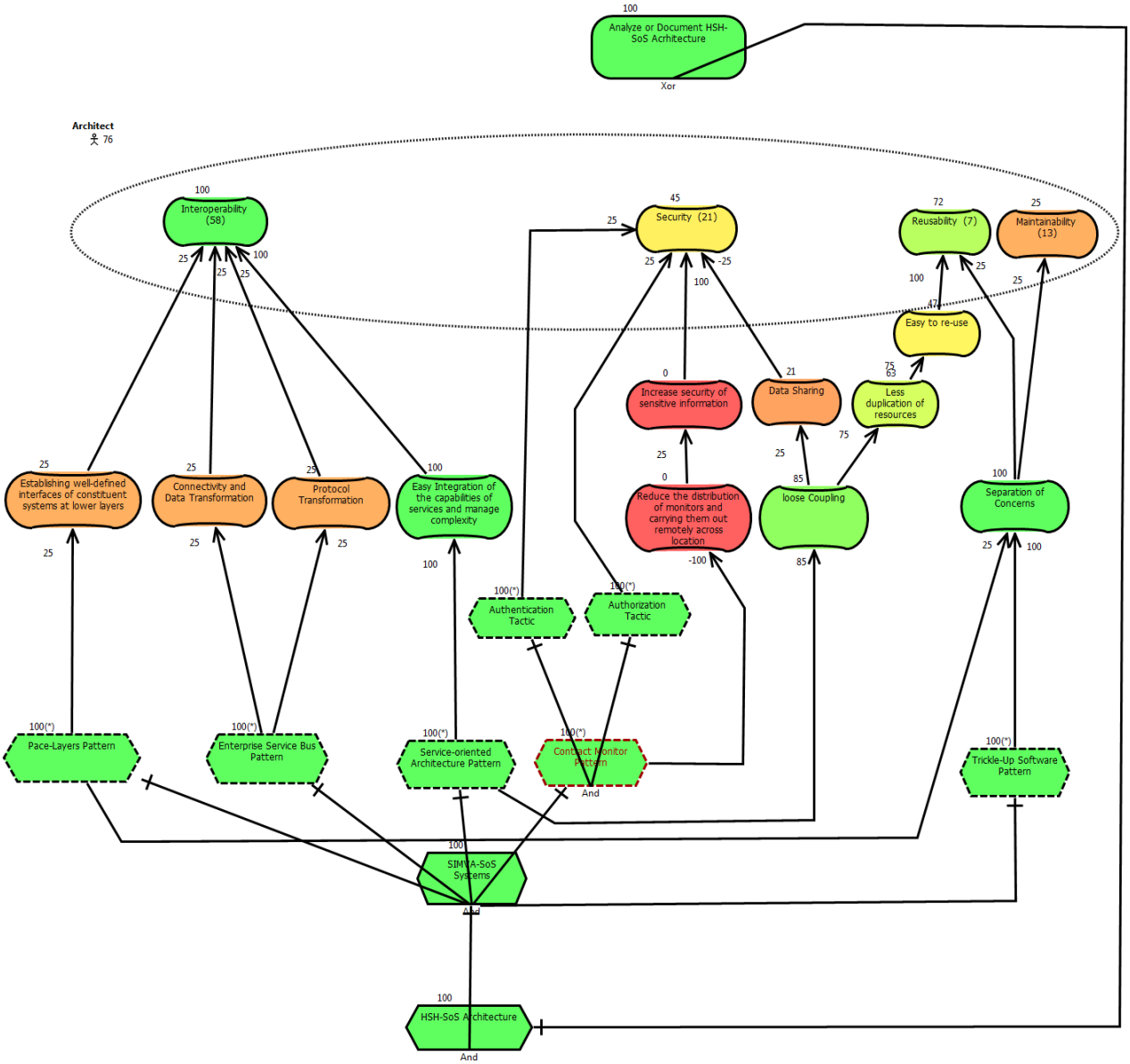


Figure 48 Evaluated GRL model where patterns/tactics of HSH-SoS are initially satisfied after running the jUCMNav propagation

7.2.3.2. Evaluate the Software Architectures

Based on the evaluation results of the GRL model from the previous step, *Interoperability* has a higher satisfaction level **(100)** compared to the other requirements. This is due to the implementation of the ***Service-oriented Architecture*** pattern which promotes *Interoperability* among distributed systems to accomplish business functionalities

(services) through the ease of integration of their capabilities. Additionally, the implementation of the ***Pace-Layers*** and ***Enterprise Service Bus*** patterns also helps improve interoperability. The ***Enterprise Service Bus*** pattern provides connectivity, data transformation for semantic and syntactic interoperability and protocol transformation [56], while the ***Pace-layers*** pattern favors well-defined interfaces of constituent systems.

The maintainability requirement has a lower satisfaction level in the HSH-SoS architecture **(25)**. *Maintainability* is improved by only one design decision (separation of concerns with contribution value **(25)**). This decision is derived from implementing the ***Trickle-Up Softwar***e pattern. *Reusability* has satisfaction level **(72)**, and the *Security* has satisfaction level **(45)**. The overall satisfaction value of the architect is **(76)**.

7.3. Case Study 3: Build-Automation Systems

This case study is about two build-automation systems, Gradle [115] and Maven [116]. They are both used to build, test, deploy, and deliver software. Maven was released in 2004. It uses Extensible Markup Language (XML) [117] as the format for build scripts. Gradle was released in 2012. Google adopted Gradle as the default build tool for the Android OS. Gradle uses a Domain-Specific Language (DSL) [118] based on Groovy [119] (one Java-Virtual Machine (JVM) language). As a result, Gradle builds script tends to be much shorter and more precise than those written for Maven. Various resources and websites report that Gradle is faster than Maven [120][121][122][123][124][125]. In this case study, I applied our approach to the implementations of Maven and Gradle. Our objective is to be able to compare the evaluation obtained with our approach in regard to performance, to these reports and identify aspects of the architecture justifying a better performance of Gradle compared to Maven.

In the following, I introduce the two tools Gradle and Maven.

7.3.1. Gradle

Gradle [115] is an open-source build automation tool focused on flexibility and performance. Gradle build scripts are written using a Groovy or Kotlin DSL. Gradle is

modelled in a way that is customizable and extensible in the most fundamental ways. It completes tasks quickly by reusing outputs from previous executions, processing only inputs that changed, and executing tasks in parallel. Gradle is the official build tool for Android and comes with support for several languages and technologies. The interested reader can refer to [115] for more details.

7.3.2. Maven

Maven [116] is an open-source build automation tool. It is a software project management and comprehension tool. Maven is based on the concept of a project object model (POM) [126], Maven can manage a project's build, reporting and documentation from a central piece of information. It allows a developer to comprehend the complete state of a development effort in the shortest time period. The interested reader can refer to [116] for more details.

Note that, since the patterns and tactics in the context of build-automation systems are the same as the ones in the context of big data systems, I do not need to model the determined patterns and tactics for this case study. I already modeled them when I applied our approach to the first case study (cyber fusion center).

7.3.3. Determining the Patterns and Tactics Implemented in Software Architecture

I apply *Archie* to the two systems Gradle and Maven to determine their patterns and tactics, as shown in the following Sections.

7.3.3.1. Results with Gradle and Maven

I applied *Archie* to the Gradle and Maven and I got a set of candidate patterns and tactics as shown in Tables 45 and 46.

Table 45 Analysis of applying Archie to Gradle

#	Tactics	Number of Trained in Keywords	Number of Detected Keywords	Frequency of the Detected Terms	Frequency of Tactic	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	3	1	0	Yes	FP	<=0.4
2	**Heartbeat**	13	0	0	0	NO	NA	-

3	**Ping/Echo**	10	1	4	2	Yes	TP	<=0.4
4	**Exception Handling**	5	4	32	3	Yes	TP	<=0.3
5	**Authenticate**	10	2	74	27	Yes	TP	<=0.5
6	**Time Stamp**	4	2	153	6	Yes	TP	<=1
7	**Resource Pooling**	10	3	16	3	Yes	TP	<=0.6
8	**Audit Trail**	10	0	0	0	NO	NA	-
9	**PBAC**	10	2	15	0	Yes	FP	<=0.3
10	**RBAC**	10	5	61	2	Yes	TP	<=0.4
11	**Resource Scheduling**	10	7	36	4	Yes	TP	<=0.3
12	**Session Management**	10	4	9	4	Yes	TP	<=0.5
13	**Load Balancing**	10	3	1	0	Yes	FP	<=0.4
14	**Restart**	4	1	14	2	Yes	TP	<=1
15	**Time-Out**	3	1	420	15	Yes	TP	<=1
16	**Cancel**	1	1	112	12	Yes	TP	<=1
17	**Introduce Concurrency**	10	1	3	1	Yes	TP	<=0.4
18	**Checkpoint**	12	1	1	0	Yes	FP	<=05
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained in Keywords**	**Number of Detected Keywords**	**Frequency of the Detected Terms**	**Frequency of Pattern**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	5	2	Yes	TP	<=0.9
2	**Broker**	10	0	0	0	NO	NA	-
3	**Observer/Publish-Subscribe**	4	3	233	13	Yes	TP	<=0.3
4	**Pipes and Filters**	3	4	281	11	Yes	TP	<=1
5	**Shared-Repository**	4	0	0	0	NO	NA	-

Table 46 Analysis of applying Archie to Maven

#	Tactics	Number of Trained in Keywords	Number of Detected Keywords	Frequency of the Detected Terms	Frequency of Tactic	Detected by Archie	TP/ FP	Threshold
1	**Kerberos**	10	0	0	0	NO	NA	-

2	**Heartbeat**	13	0	0	0	NO	NA	-
3	**Ping/Echo**	10	0	0	0	NO	NA	-
4	**Exception Handling**	5	3	8	2	Yes	TP	<=0.3
5	**Authenticate**	10	3	16	5	Yes	TP	<=0.4
6	**Time Stamp**	4	2	21	6	Yes	TP	<=1
7	**Resource Pooling**	10	0	0	0	NO	NA	-
8	**Audit Trail**	10	0	0	0	NO	NA	-
9	**PBAC**	10	3	36	1	Yes	FP	<=0.3
10	**RBAC**	10	4	210	7	Yes	FP	<=0.4
11	**Resource Scheduling**	10	3	3	1	Yes	FP	<=0.2
12	**Session Management**	10	2	24	3	Yes	FP	<=0.2
13	**Load Balancing**	10	0	0	0	NO	NA	-
14	**Restart**	4	0	0	0	NO	NA	-
15	**Time-Out**	3	1	28	3	Yes	TP	<=0.7
16	**Cancel**	1	1	121	15	Yes	TP	<=1
17	**Introduce Concurrency**	10	1	2	0	Yes	FP	<=0.4
18	**Checkpoint**	12	0	0	0	NO	NA	-
19	**Retry**	3	0	0	0	NO	NA	-
#	**Patterns**	**Number of Trained in Keywords**	**Number of Detected Keywords**	**Frequency of the Detected Terms**	**Frequency of Pattern**	**Detected by Archie**	**TP/ FP**	**Threshold**
1	**Layers**	4	1	1	1	Yes	TP	<=0.9
2	**Broker**	10	0	0	0	NO	NA	<=1
3	**Observer/Publish-Subscribe**	4	1	8	2	Yes	TP	<=0.2
4	**Pipes and Filters**	3	2	67	11	Yes	TP	<=0.5
5	**Shared-Repository**	4	0	0	0	NO	NA	-

Figures 49 and 50 show a sample of the detected patterns and tactics when I run the Archie tool on Gradle's and Maven's source code.

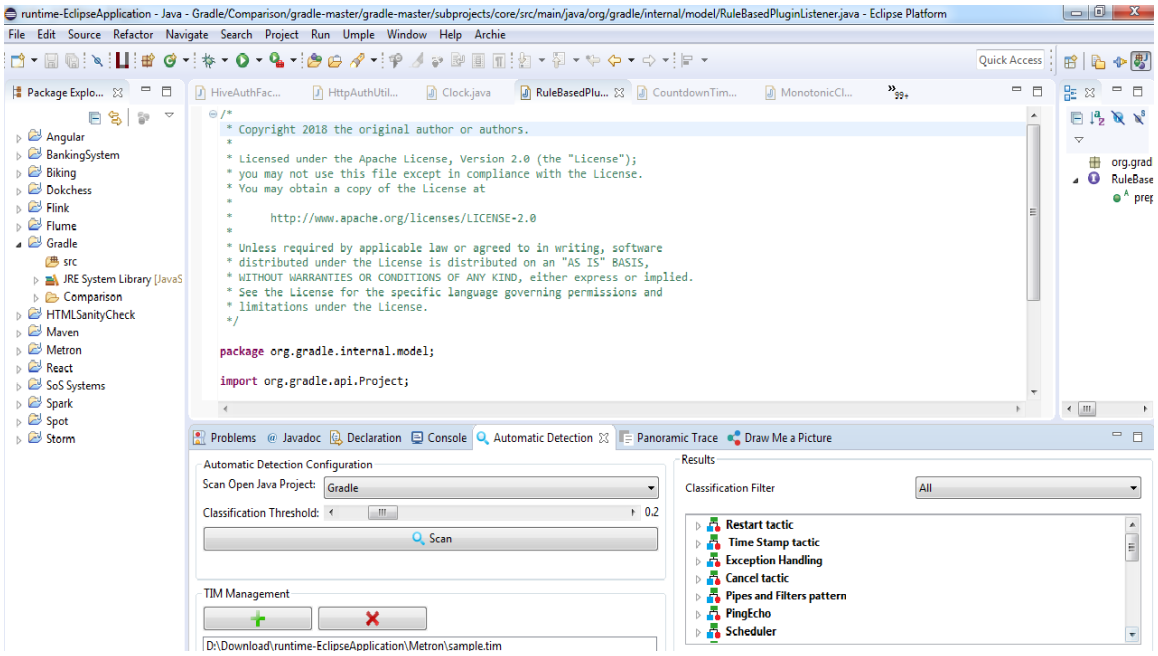


Figure 49 Sample of the detected patterns/tactics for Gradle

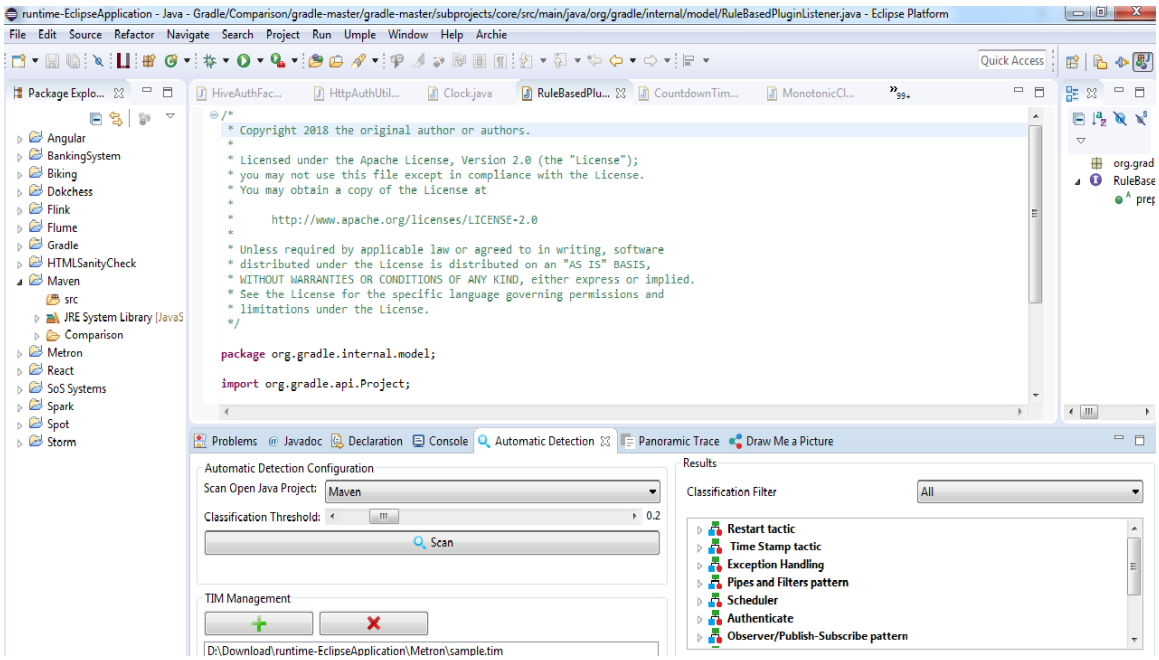
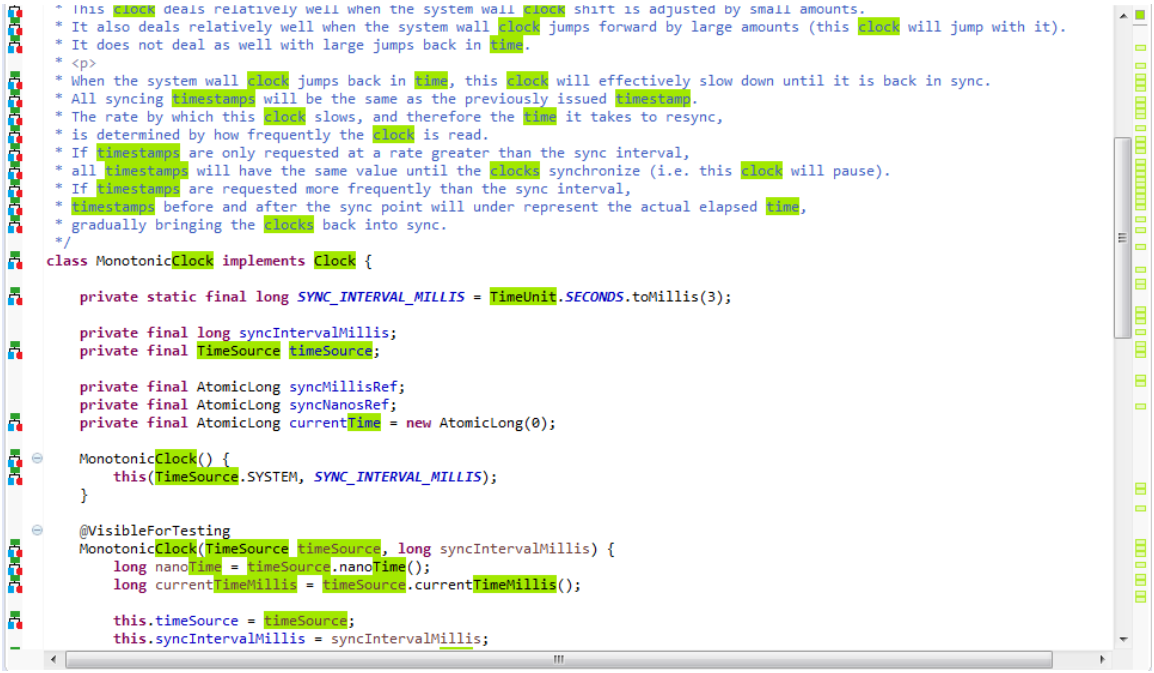


Figure 50 Sample of the detected patterns/tactics for Maven

I then validated the results of *Archie* for Gradle and Maven by looking for the occurrences of those patterns/tactics, detected by *Archie*, manually in the source code/documentation/websites of Gradle and Maven. For example, Figure 51 shows the

source code where the *"Time Stamp"* tactic is detected in Gradle and its related terms. Figure 52 shows the source code where the *"Authentication"* tactic is detected in Maven and its related terms.



```
 * This clock deals relatively well when the system wall clock shift is adjusted by small amounts.
 * It also deals relatively well when the system wall clock jumps forward by large amounts (this clock will jump with it).
 * It does not deal as well with large jumps back in time.
 * <p>
 * When the system wall clock jumps back in time, this clock will effectively slow down until it is back in sync.
 * All syncing timestamps will be the same as the previously issued timestamp.
 * The rate by which this clock slows, and therefore the time it takes to resync,
 * is determined by how frequently the clock is read.
 * If timestamps are only requested at a rate greater than the sync interval,
 * all timestamps will have the same value until the clocks synchronize (i.e. this clock will pause).
 * If timestamps are requested more frequently than the sync interval,
 * timestamps before and after the sync point will under represent the actual elapsed time,
 * gradually bringing the clocks back into sync.
 */
class MonotonicClock implements Clock {

    private static final long SYNC_INTERVAL_MILLIS = TimeUnit.SECONDS.toMillis(3);

    private final long syncIntervalMillis;
    private final TimeSource timeSource;

    private final AtomicLong syncMillisRef;
    private final AtomicLong syncNanosRef;
    private final AtomicLong currentTime = new AtomicLong(0);

    MonotonicClock() {
        this(TimeSource.SYSTEM, SYNC_INTERVAL_MILLIS);
    }

    @VisibleForTesting
    MonotonicClock(TimeSource timeSource, long syncIntervalMillis) {
        long nanoTime = timeSource.nanoTime();
        long currentTimeMillis = timeSource.currentTimeMillis();

        this.timeSource = timeSource;
        this.syncIntervalMillis = syncIntervalMillis;
```

Figure 51 Source code where Time Stamp tactic is detected in Gradle and its related terms

```
package org.apache.maven.artifact.repository;

 * Licensed to the Apache Software Foundation (ASF) under one

/**
 * Authentication
 */
public class Authentication
{

    private String privateKey;

    private String passphrase;

    public Authentication( String userName, String password )
    {
        this.username = userName;
        this.password = password;
    }

    /**
     * Username used to login to the host
     */
    private String username;

    /**
     * Password associated with the login
     */
    private String password;

    /**
     * Get the user's password which is used when connecting to the repository.
     *
     * @return password of user
     */
    public String getPassword()
```

Figure 52 Source code where Authentication tactic is detected in Maven and its related terms

I calculate the numbers of the True Positives (TP), the False Positives (FP) in Gradle and Maven. The numbers of the TP, FP, and accordingly the precision for Gradle and Maven are shown in Table 47.

Table 47 Numbers of TP, FP, and Precision for Gradle and Maven

Tool	TP	FP	Precision
Gradle	15	4	0.79
Maven	8	5	0.62

7.3.1.3. Overlaps between Patterns and Tactics

In this step, I determine the overlaps between patterns and tactics of both Gradle and Maven tools, as shown in Tables 48 and 49.

Table 48 Results of the overlaps in Gradle

Pattern	Java Class(s)	Tactic(s)	Overlaps
Pipes and Filter	FilterChain.java	Exception Handling	1
	LineFilter.java	Close, Resource Scheduling	1
Observer/Publish-Subscribe	BuildCacheControllerFactory.java	Timestamp	1
	Connection.java	Timeout	1
Layers	-	-	-

Table 49 Results of the overlaps in Maven

Pattern	Java Class(s)	Tactic(s)	Overlaps
Pipes and Filter	-	-	-
Observer/Publish-Subscribe	MavenCli.java	Timeout	1
		Exception Handling	1
Layers	-	-	-

As I can see in Tables 48 and 49, the patterns ***Pipe and Filters*** and ***Observer/Publish-Subscribe*** have been overlapped with only the tactics *"Timeout", "Exception Handling", "Timestamp", "Close", and "Resource Scheduling"*. Each tactic has a single class overlapped with the patterns ***Pipe and Filters*** and ***Observer/Publish-Subscribe***.

7.3.4. Modelling Software Architectures in terms of their Implemented Patterns and Tactics

I modelled Gradle and Maven in terms of their implemented patterns and tactics

7.3.4.1. Identify the Most Architecturally Significant NFRs to be Considered in the Design of Software Architectures in the Determined Context

I consider only the *Performance* as the most architecturally significant NFR for this case study. Our objective with this case study is to compare the evaluation obtained with our approach in regard to performance to other reports and to identify aspects of the architecture justifying a better performance of Gradle compared to Maven.

7.3.4.2. Calculate the importance values of the considered NFRs

Since I only want to compare the performance of both Gradle and Maven tools, the importance value of *Performance* should be set to **(100)**.

7.3.4.3. Modelling Software Architecture in terms of their Implemented Patterns and Tactics

I modeled the two tools Gradle and Maven in terms of their implemented patterns and tactics in regard to performance as shown in Figure 53.

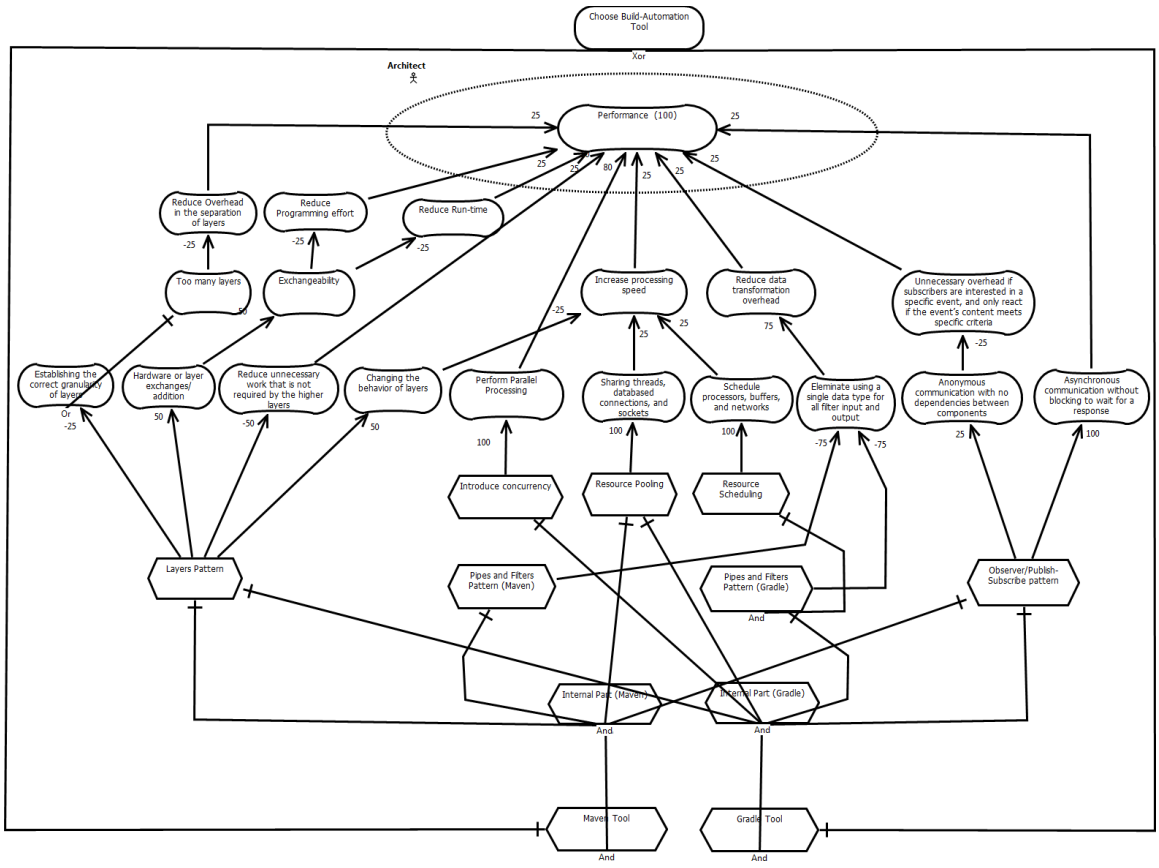


Figure 53 GRL model of Maven and Gradle in terms of Performance requirement

As can be noted in Figure 53, the high-level goal of the systems is shown at the top of the model connected to alternative candidate tools at the bottom of the model. The high-level goal is to compare the performance of both Gradle and Maven tools. The ***Pipes and Filters*** pattern has a negative contribution of type **(-75)** with *Performance*. This is justified by that

the ***Pipes and Filters*** pattern uses a single data type for all filter input and output to achieve the highest flexibility results in data conversion overheads [84]. The "*Resource Pooling"* and the *"Resource Scheduling"* tactics have positive contribution values of type **(100)** with the *Performance* requirement. The *"Resource Pooling"* allows sharing threads, database connections, and sockets. While the *"Resource Scheduling"* allows to schedule processors, buffers, and networks. The ***Observer/Publish-Subscribe*** pattern has a positive contribution value of type **(100)** with *Performance*, and at the same time, it has a negative contribution value of type **(-25)** with *Performance*. It provides asynchronous communication without blocking to wait for a response and allows components in an application to coordinate their computation anonymously without introducing explicit dependencies to one another: they are unaware and independent of each other's location and identity since they only send and receive events about changes of their state and/or the changed state itself. However, anonymous communication can cause unnecessary overhead if subscribers are interested in a specific type of event, and will only react if the event's content meets specific criteria. The ***Layers*** pattern has negative contribution values with *Performance* of type **(-50)** and **(-25)**. This is because of the granularity of layers and performing unnecessary work that is required by the higher layers. The *Performance* importance value is **(100)** as previously mentioned.

7.3.5. Evaluate the Software Architectures

7.3.5.1. Evaluate the Model of the Software Architectures

I evaluate the model to calculate the satisfaction levels of the NFRs (Figure 53). The evaluation is done by applying different evaluation strategies on the GRL model. Figures 54 and 55 show two different strategies before running the jUCMNav propagation and getting the satisfaction levels. The first strategy in Figure 54 models the application of Gradle where only its patterns and tactics are initially satisfied. Figure 55 models the application of Maven.

Figure 56 and 57 show the evaluation strategies, for Gradle and Maven respectively, after running the jUCMNav propagation and getting the satisfaction levels for each node in the model.

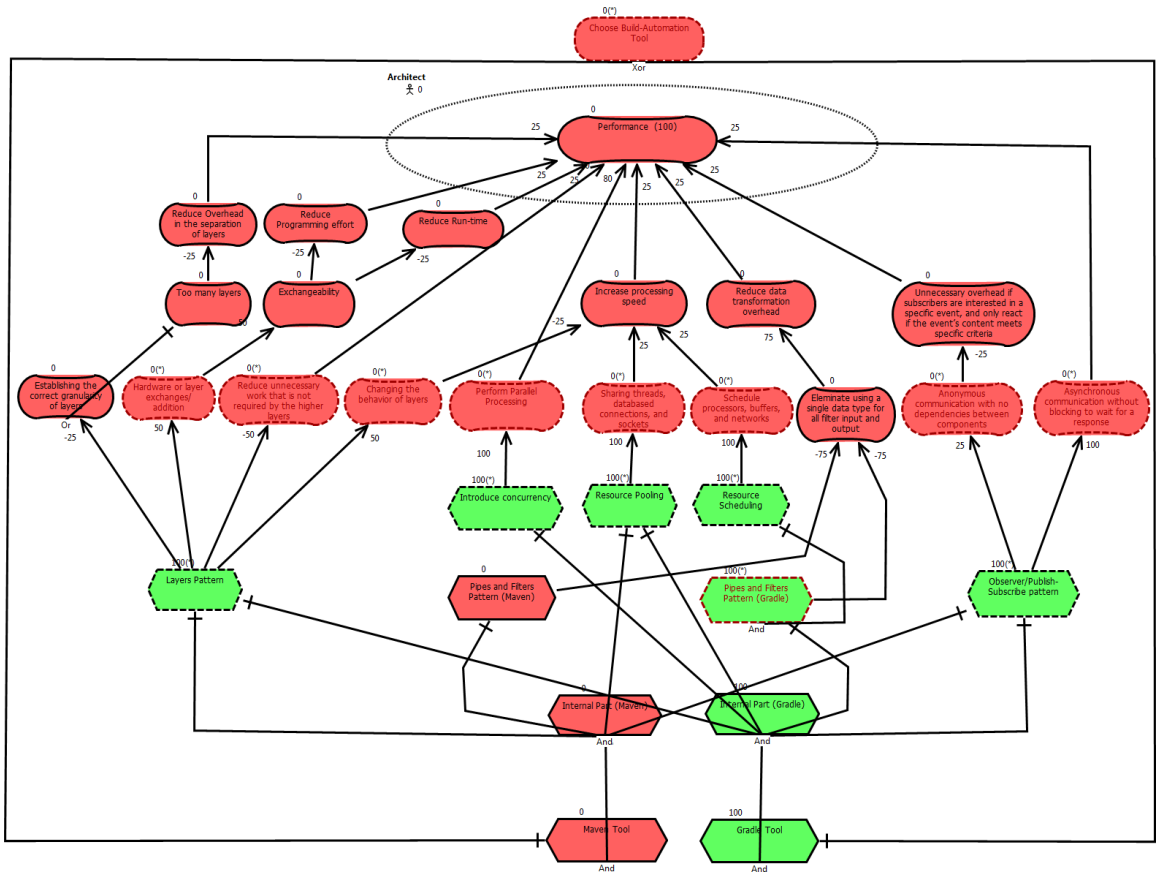


Figure 54 GRL model where patterns/tactics of Gradle are initially satisfied in terms of Performance before running the jUCMNav propagation

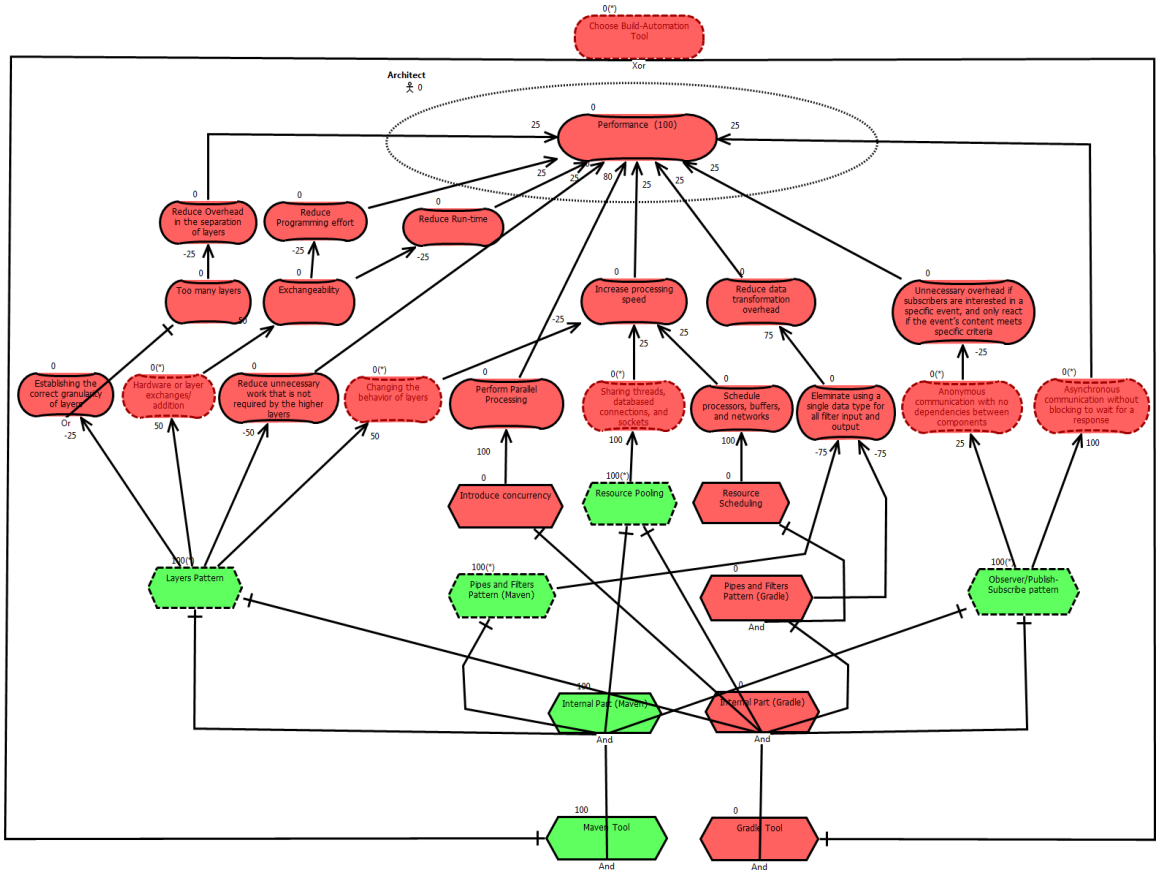


Figure 55 GRL model where patterns/tactics of Maven are initially satisfied in terms of Performance before running the jUCMNav propagation

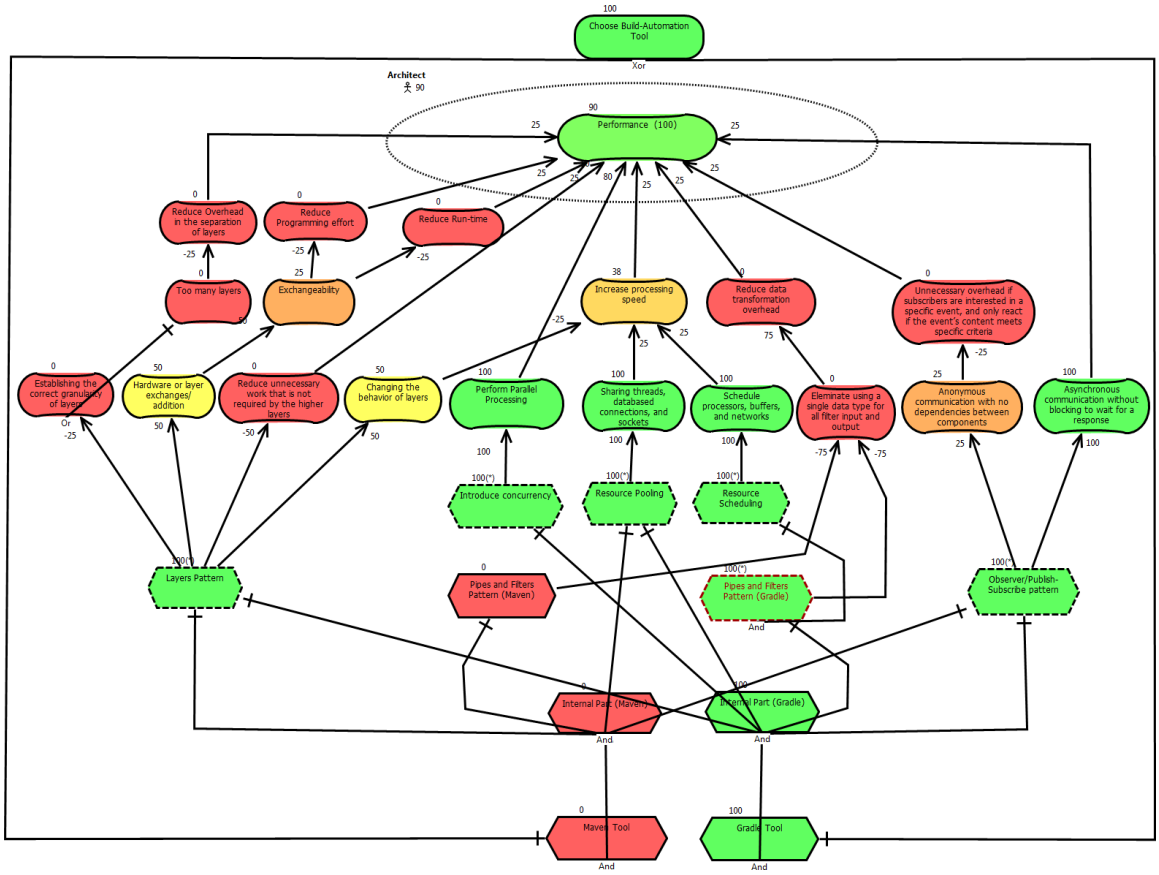


Figure 56 Strategy 1: Evaluated GRL model where patterns/tactics of Gradle are initially satisfied in terms of Performance after running the jUCMNav propagation

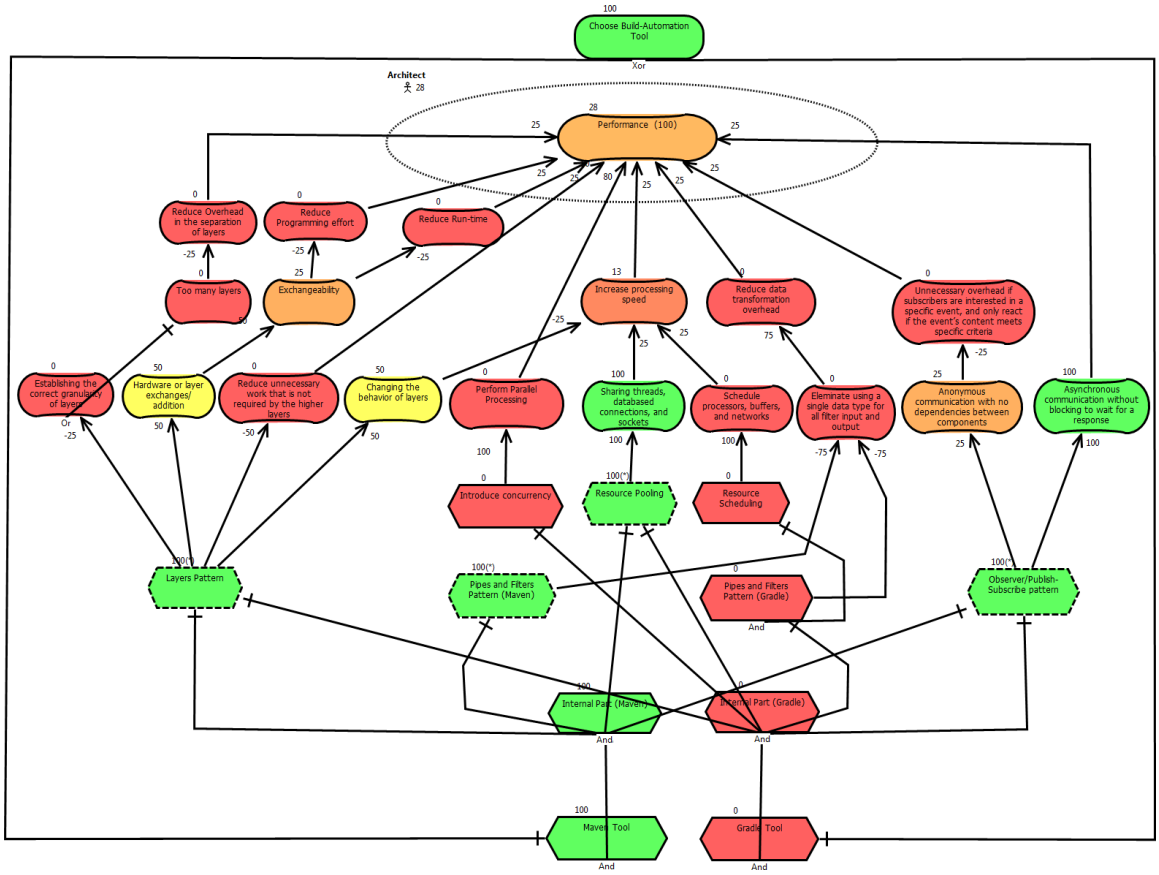


Figure 57 Strategy 2: Evaluated GRL model where patterns/tactics of Maven are initially satisfied in terms of Performance after running the jUCMNav propagation

7.3.5.2. Evaluate the Software Architectures

Based on the evaluation results of the GRL models from the previous step, I can see that the Gradle tool has a higher satisfaction level for Performance, which is **(90)**, as shown in Figure 56. This is because of the implementation of the three performance tactics *"Introduce Concurrency", "Resource Pooling",* and the *"Resource Scheduling"* in Gradle. The *"Introduce Concurrency"* tactic allows parallel execution of tasks such that the blocked time can be reduced. The *"Resource Pooling"* tactic allows sharing threads, database connections and sockets. While the *"Resource Scheduling"* tactic allows to schedule processors, buffers and networks. The ***Layers*** pattern in both Gradle and Maven causes a decrease of the satisfaction level of *Performance* for both Gradle and Maven. This is due to the granularity of layers and the execution of unnecessary work required by higher

layers. This results in increased overhead and increased run-time as well as programming effort.

Since the satisfaction level for *Performance* in Gradle is **(90)**, while in Maven is **(28)**, this confirms that Gradle has better performance (runs faster) than Maven.

7.4. Chapter Summary

In this Chapter, I presented three case studies from different contexts to validate our approach. The first case study is about the task of choosing a stream processing framework for a cyber fusion center. Three candidate frameworks were suggested: Apache Storm, Apache Flink, and Apache Spark. The three frameworks have been modeled and compared to each other based on their implemented patterns and tactics considering the importance values of the given NFRs. The second case study is based on the Health supportive Home –System of Systems (HSH-SoS) in the context of System of Systems (SoS). The architecture of the HSH-SoS has been modeled in terms of its implemented patterns and tactics. The objective is to analyze and understand the implementation and design of the HSH-SoS architecture in terms of its implemented patterns and tactics and their impact on its quality attributes. The third case study evaluates two build-automation tools: Gradle and Maven. They are both used to build, automate, test, deploy, and deliver faster software in the context of the build-automation systems. The performance of both Gradle and Maven have been modeled and compared.

Chapter 8. Validation and Evaluation

This chapter presents the validation and evaluation that was done throughout this research. Since our research method, described in Section 1.4, is an adaptation of the design science research method, the validation was not left to the end. Some level of validation was done at various stages of the research and the output was used in a feedback loop, which improved the intermediate and final outcomes. In summary, I performed the following validation tasks.

1) *Literature Review*: I compared our approach to related approaches based on several criteria.

2) *Real-World Case Studies*: I validated our approach by applying it to real-life case studies to illustrate that it can be applied in different contexts. I analyzed the results to be able to answer our research questions introduced in chapter 1. I present the details on these case studies in Chapter 7 and summarize what I learned in this chapter.

3) *Manual Checking of Source Code*: I checked the source code of software architectures by manually looking for the occurrences of patterns and tactics. This has been done for the purpose of validating the results of Archie. I determine the numbers of the True Positives (TP), False Positives (FP), and consequently, the values of precision for the detected patterns and tactics.

4) *Comparisons with Benchmark Results from the Literature*: To validate our approach, I compared the results, of applying our approach to real-world case studies, with benchmark comparisons from the literature.

8.1. Evaluation Based on the Literature Review

In Chapter 2, I conducted a literature review on four search areas:

(1) Existing approaches to connect software implementation to quality attributes. This is discussed in Section 2.2.

(2) Existing approaches to determine patterns and tactics in an implementation. This is discussed in Section 2.3.
(3) Existing techniques for modeling software in terms of their implemented patterns and tactics. This is discussed in Section 2.4.
(4) Existing approaches for evaluating software architectures. This is discussed in Section 2.5.

I investigated the novelty and feasibility of our proposed approach by seeking answers to the following questions:

- Question 1: "Are there existing approaches that connect software implementation to quality attributes?"
- Question 2: "Are there existing approaches to determine (extract) the patterns and tactics implemented by a software?"
- Question 3: "Are there existing techniques for modeling software architectures in terms of their implemented patterns and tactics?"
- Question 4: "Are there existing approaches to evaluate software architectures based on their implemented patterns/tactics"

After reviewing the papers found, I can conclude that our approach is novel and feasible. The approach is novel because I did not find any approach that connects software implementation to quality attributes based on its implemented patterns and tactics. Our approach is feasible because there is a number of works showing that patterns and tactics are linked to quality attributes (both positively and negatively) and that they can be extracted from code.

As illustrated in Table 50 (already shown as Table 3 in the literature review), only the approach proposed in this thesis connects software implementation to quality attributes using both patterns and tactics and is supported by a model. It is also the only approach which extract both architectural patterns and tactics semi-automatically.

Table 50 Comparisons between the related works and our approach

Approach	Category	Consider Patterns	Consider Tactics	Extraction method of patterns/tactics	The degree of automation of the identification of patterns and tactics	Consider the modelling of the architecture
SAAM [4]	Scenario-based	NO	NO	----------	----------	NO
ATAM [4]	Scenario-based	NO	NO	----------	----------	NO
CBAM [4][66]	Scenario-based	NO	NO	----------	----------	NO
ALMA [67]	Scenario-based	NO	NO	----------	----------	NO
FAAM [68]	Scenario-based	NO	NO	----------	----------	NO
LQN [7]	Simulation-based, Mathematical modeling	NO	NO	----------	----------	Yes
ARGUS-I [63]	Simulation-based	NO	NO	----------	----------	NO
SAM [64]	Simulation-based	NO	NO	----------	----------	NO
EBAE [61]	Experience-based	NO	NO	----------	----------	NO
ABAS [62]	Experience-based	NO	NO	----------	----------	NO
SPE [65]	Mathematical modeling, Simulation-based,	NO	NO	----------	----------	Yes
Cervantes et al. [1]	Documentation-based	YES	NO	Attribute-Driven Design (ADD)	Non-automated	NO
Johnson [16]	----------	Yes	NO	----------	----------	NO

Mirakhorli and Huang [11] [12] [13]	----------	NO	Yes	Archie tool	Semi-Automated	NO
Aguiar and David [70]	----------	Yes	NO	----------	----------	NO
Beck and Johnson [18]	Documentation-based	Yes	NO	problem statement of an architecture	Non-automated	NO
Heesch, Avgeriou, and Hilliard [71]	----------	Yes	NO	----------	----------	NO
Sena et al. [72]	NA	Yes	NO	Literature review	Non-automated	NO
Sena et al. [73]	NA	Yes	NO	Literature review	Non-automated	NO
Ryoo et al. [74]	Documentation-based	NO	Yes	Interviews	Non-automated	NO
Meusel et al [17]	Documentation-based	Yes	NO	Pattern instantiation	Non-automated	NO
Zhu [69]	Scenario-based	Yes	NO	Manually	Non-automated	NO
Our approach (SAEPT)	**Patterns-Tactics based, Modeling-based**	**Yes**	**Yes**	**Archie tool and manual search**	**Semi-automated**	**Yes**

I rated the degree of automation of the identification of patterns and tactics in our approach as "semi-automated" though a large part of the identification is performed automatically with Archie. This is due to the fact that I manually verify the effectiveness of the detected patterns/tactics and eliminate false positives before the modelling stage. This is necessary due to the level of precision of our adaptation of *Archie* for pattern detection. A more precise tool would make the process completely automated.

8.2. Evaluation Using Case Studies

The application of our approach to real-world case studies helped us detect shortcomings and come up with ideas for potential future improvements. The details on these case studies are available in Chapter 7. I present a summary of what I learned from these case studies here.

8.2.1. Cyber Fusion Center Case Study

In Section 7.1, I presented an application of our approach to aid the selection of a software framework for a cyber fusion center. I modeled the three candidate frameworks Storm, Flink, and Spark in term of their implemented patterns and tactics. This allowed a comparison of these frameworks based on their implemented patterns and tactics considering the importance values of NFRs. The evaluation provided a rationale for architects tasked to choose among these frameworks.

This case study allowed us to realize that choosing a framework based on its implemented patterns and tactics considering the importance values of the NFRs may not be sufficient. Additional criteria are needed including context, stability, maturity, and community of a framework. Based on that realization, I intent to take additional criteria into consideration in our future work.

I also realized that our approach can be applied to more than two frameworks in the same context. Our evidence is that I first applied the approach to only two data streaming frameworks Apache Storm and Apache Flink. Then, I extended it to include another data streaming framework, Spark. My approach worked properly with both cases. This helped me to come up with a future work of applying my approach to compare more than three frameworks.

Finally, I realized that the model of the frameworks can help an architect not only to compare candidate frameworks, but also to understand the architecture of the frameworks. The models created present frameworks in terms of their implemented patterns and tactics and the satisfaction level of each requirement by these patterns and tactics. They are based on recognized documentation. This motivated us to conduct the HSH-SoS case study discussed in Section 7.2 and summarized in the next section, to concentrate on this aspect.

8.2.2. Healthcare Supportive Home (HSH)-System of Systems (SoS)

I present the HSH-SoS case study in Section 7.2. The application of our approach to the HSH-SoS confirmed that our approach can be used not only to compare several candidate architectures, but to provide a rationale about a single architecture. The provided rationale is in terms of the implemented patterns and tactics and the impact as well as tradeoffs on requirements. The HSH-SoS also demonstrated that our approach can be applied to software from different contexts. As future work, I plan on continuing to apply to more case studies in different contexts.

8.2.3. Build-Automation Systems

In Section 7.3, I presented an application of our approach to two build-automation tools Gradle and Maven. Our goal is, to provide some validation to the main hypothesis of our research by comparing our inferred quality attributes (i.e. performance) to benchmark values.

I modeled the two tools in term of their implemented patterns and tactics. This allowed a comparison of the inferred performance NFR of the two tools based on their implementation, to benchmark values.

I realized that the fact that Gradle is faster than Maven comes from implementing the three performance tactics *"Introduce Concurrency", "Resource Pooling",* and *"Resource Scheduling"*. This helped us to come up with a future work of comparing the two tools based on different NFRs (i.e. *availability, reliability,* and *flexibility*).

I also realized that the model of the two tools can be used not only to compare the tools with benchmark values, but also to understand the architecture of the tools in terms of the performance. The models created present the two tools in terms of their implemented patterns and tactics and the satisfaction level of the performance by these patterns and tactics.

Finally, I realize that comparing these tools in terms of the performance based on their implemented patterns and tactics may not be sufficient. As previously mentioned, additional criteria are needed including the used programming language, the cost, and memory requirements of a tool. Based on that realization, I intent to take these additional criteria into consideration in our future work.

8.3. Evaluation Based on the Manual Checking of Source Codes

In Chapter 4, I adapted *Archie* to determine the implemented patterns and tactics of a framework. I then validated the results of *Archie* by looking for the occurrences of the patterns and tactics in the source code of a software architecture. I calculated the True Positives (TP) and False Positives (FP), and derived the precision of Archie for each case study as presented in Chapters 4 and 7, and as shown in Table 51. As can be noticed in Table 51, in the first case study (cyber fusion center), the True Positives (TP), False Positives (FP), and precision respectively for Storm are **(19)**, **(2)**, and **(0.90)**, while they are **(18)**, **(4)**, **(0.82)** for Flink and they are **(15)**, **(4)**, **(0.79)** for Spark.

In the second case study (HSH-SoS), the True Positives (TP), False Positives (FP), and precision values are **(16)**, **(4)**, and **(0.80)** respectively. In the third case study (Gradle and Maven), **(15)**, **(4)**, and **(0.79)** are the True Positives (TP), False Positives (FP), and precision values for Gradle, while they are **(8)**, **(5)**, and **(0.62)** for Maven.

The average of the precision of *Archie* for all the case studies in this thesis is 0.79. The number of False Positives is enough to falsify our results. I therefore, manually validate *Archie* results and only consider the True Positives for the models.

I also do not determine the numbers of the False Negatives (FN). Consequently, there could be architectural tactics and patterns present in the code but not accounted in our case studies. In order to determine False Negatives (and True Negatives) a detailed inspection of the code that must account for the various ways architectural patterns/tactics can be implemented is needed. I did not find this feasible given the size of the analyzed systems.

One of the potential ways to improve the precision of detected architectural patterns and tactics is to use *Archie* only for tactics and another tool for patterns. *Archie* is originally developed to detect tactics. As a potential future work, I will investigate this approach with the objective of making architectural patterns and tactics detection completely automated.

Table 51 Summary of the numbers of TP, FP, and Precision for all case studies

Case Study	Software Architecture	TP	FP	Precision
Cyber Fusion Center	Storm	19	2	0.90
	Flink	18	4	0.82
	Spark	15	4	0.79
HSH-SoS	HSH-SoS architecture	16	4	0.80
Build-Automation Systems	Gradle	15	4	0.79
	Maven	8	5	0.62
Average				**0.80**

8.4. Evaluation Based on the Comparisons with Benchmark Results from the Literature

I compare our case studies results with benchmark results from the literature.

8.4.1. Cyber Fusion Center

I compare the inferred quality attributes (i.e. reliability, availability, and performance) with benchmark comparison results, such as [114][127][128][129][130][131][132]. Most of these benchmarks ([127][129][131][132]) only compare the three frameworks Storm, Flink, and Spark on the performance in terms of **throughput** and **latency**. Other benchmarks [114][130] evaluate the frameworks based on performance in terms of **processing time**. Note that, in this thesis, I consider the performance only in terms of the **processing time**. Benchmarks [114][128] evaluate the frameworks in term of fault tolerance. The study [114] compares the three frameworks based on both the performance (i.e. **processing time**) and the fault tolerance (*reliability* and *availability*). I compare our results with these benchmarks in terms of *reliability, availability,* and *performance* (i.e. **processing time**) only. I consider [114] results on fault tolerance for the comparison of the frameworks on reliability and availability.

Inoubli et al. [114] conducted an experimental survey on big data frameworks. They compared big data frameworks such as Hadoop, Storm, Flink, and Spark. Inoubli et al. conducted an extensive comparative study between these frameworks based on criteria such as fault tolerance, the provision of machine learning library, the programming model, the programming languages supported, and whether a framework allows iterative processing. They also performed an extensive set of experiments to highlight the strengths and weaknesses of these frameworks. Their analysis covered performance.

Our results are compatible with Inoubli's. For example, Inoubli et al. [114] showed that both Storm and Flink use the *"Checkpoint"* tactic for fault tolerance, while Spark uses recovery techniques. Lopez et al. in [128] also mentioned that Flink and Storm use the "*Checkpoint"* tactic, while Spark uses parallel recovery as a failure mechanism. The GRL model in Figure 31 in Section 7.1.5.3 shows that both Storm and Flink implement the *"Checkpoint"* tactic to record consistent states and can revert back to these states if necessary, while Spark uses the *"Active Redundancy"* tactic for recovery, preparation, and recovery from errors.

Inoubli also showed that Spark is the fastest framework in terms of the **processing time** compared to Storm and Flink. Chintapalli et al. at Yahoo in [127] and Hesse and Lorenz in [129] compared the three frameworks based on performance in terms of latency and throughput. They also showed that Flink and Storm have very similar performance, while Spark Streaming has much higher latency and is expected to handle a much higher throughput as shown in Figure 58. This confirms that Spark has better performance than Storm and Flink. Kaur et al. [130] also evaluated the performance (i.e. **processing time**) of only both Flink and Spark, based on E-commerce data from the Amazon website. They found the average time for processing data by using Flink to be 240.3seconds, while this decreased for Spark to 60.4seconds. Therefore, the performance of Spark was better than that of Flink, by approximately 179.5%. This is compatible with our results in Section 7.1.6.2, which showed that the satisfaction level of *Performance* for Spark is **(76)** while it is **(43)** for both Storm and Flink. This is because of the implementation of the ***Shared-Repository*** pattern in Spark. This pattern supports the execution of tasks influenced by the number of processors. It also supports read/write operations on RAM, rather than disk, as shown in the GRL model in Figure 42. This confirms our finding that Spark is the fastest

framework, while Storm and Flink are quite similar in terms of the data processing speed, (see Figures 41 and 42 in Section 7.1.6.1).

Inoubli also mentioned that Flink and Storm share characteristics with Spark. They implement common patterns such as the Layers and Broker patterns and tactics such as *"Resource Pooling"* and *"Resource Scheduling"*. Therefore, our results are compatible with Inoubli et al.'s.

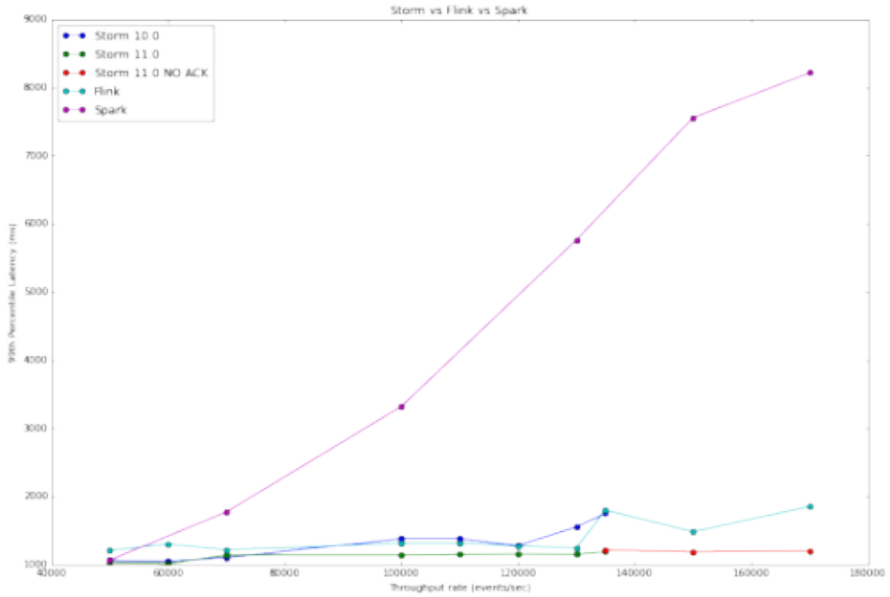


Figure 58 Throughput rate of Spark, Storm, and Flink. Adapted from [127]

Table 52 summarizes the comparisons' results for this case study.

Table 52 Comparisons with Inoubli's results [114]

Inoubli's Results	Our Results
Both Storm and Flink use the *"Checkpoint"* tactic for fault tolerance, while Spark uses recovery techniques	Figure 31 in Section 7.1.5.3 shows that both Storm and Flink implement the *"Checkpoint"* tactic to record consistent states and have a path to roll back to them if necessary, while Spark uses the *"Active Redundancy"* tactic for recovery, preparation, and repair of errors.

Spark is the fastest framework in terms of processing time compared to Storm and Flink	Our results in Section 7.1.6.2 show that the satisfaction level of the Performance for Spark is **(76)** while it is **(43)** for both Storm and Flink
Flink and Storm share similarities and characteristics with Spark	Our results in Section 7.1.5.3 show that Flink, Storm, and Spark implement the same patterns, such as the ***Layers*** and ***Broker*** patterns and the same tactics, such as *"Resource Pooling"* and *"Resource Scheduling".*

8.4.2. Healthcare Supportive Home (HSH)-System of Systems (SoS)

Our results for the HSH-SoS regarding the rationale for the architecture, are compatible with what has been described by Garces in [106]. For example, Garces mentioned that "the ***Trickle-Up*** pattern is applied in the information architecture layer. It allows to reuse data and information stored in patient records and engines' repositories, to establish patient's health situation". Our model in Figure 48 (shown in Section 7.2.3.1), shows that ***Trickle-Up*** pattern, is in fact, implemented in the HSH-SoS architecture and supports both reusability and maintainability.

Garces also mentioned the ***Contract Monitor*** pattern in the HSH-SoS architecture. It serves to verify contracts and authorize constituent systems services. In the GRL model in Figure 48, I show the Contract Monitor pattern as one of the implemented patterns of the HSH-SoS. The security tactics "*Authentication*" and "*Authorization*" are also implemented to help satisfy security requirements.

The ***Pace-Layer*** is a pattern implemented in the lower layer (system management layer) of the HSH-SoS architecture to establish well-defined interfaces of constituent systems according to Garces [106]. In Figure 48, the ***Pace-Layer*** pattern is presented as one of the implemented patterns of HSH-SoS in supports of interoperability (establish well-defined interfaces of constituent systems at lower layers).

The ***Enterprise Service Bus*** pattern is implemented in the HSH-SoS architecture for routing and transportation of services messages and events, as observed by Garces [106]. Our model in Figure 48 shows that the ***Enterprise Service Bus*** pattern supports

connectivity as well as data and protocol transformation. This helps achieve interoperability in the architecture.

Garces also mentioned the SOA pattern in the HSH-SoS architecture. As shown in Figure 48, the SOA pattern is one of the implemented patterns in the HSH-SoS architecture and it supports interoperability, security, and reusability.

Figure 59 shows the overview of the HSH-SoS architecture.

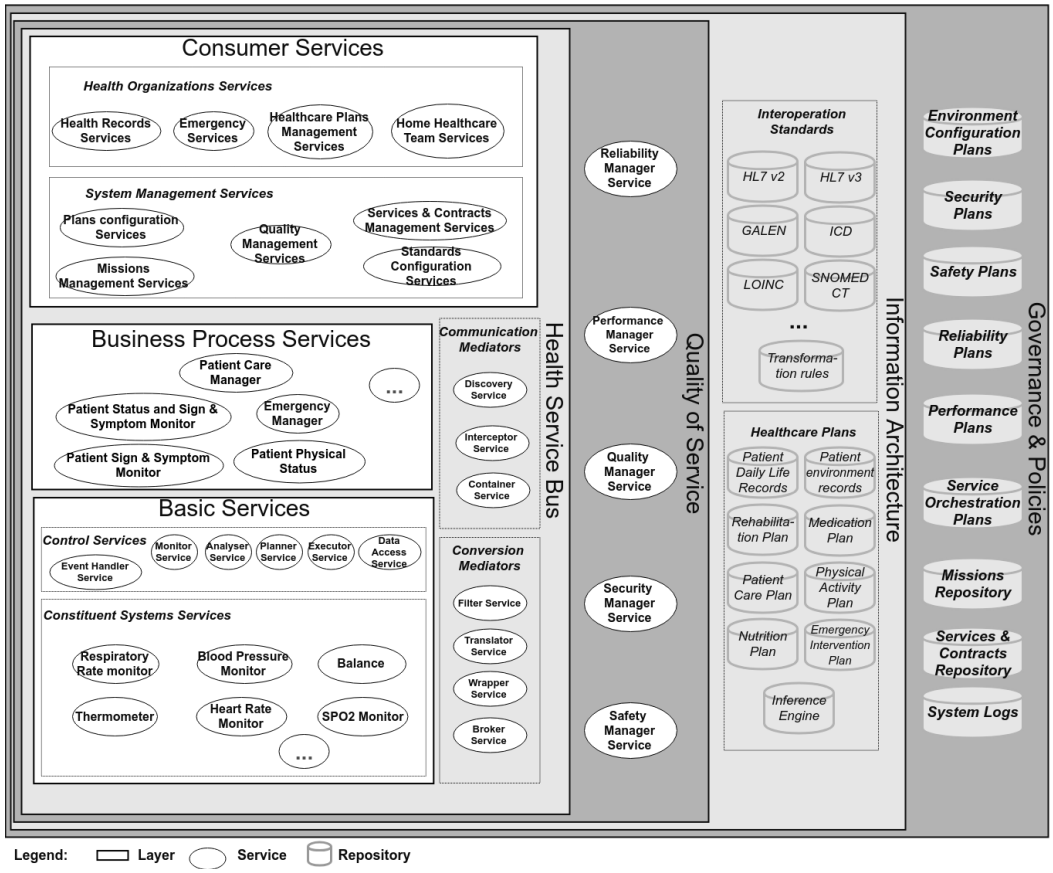


Figure 59 HSH-SoS architecture overview. Adapted from [106]

Table 53 summarizes the comparisons' results for this case study.

Table 53 Comparisons with Garces's results [106]

Garces's Results	Our Results
Trickle-Up pattern used to reuse the data and information	Our model in Figure 48 shows that ***Trickle-Up*** pattern is implemented in the HSH-SoS architecture and it supports both reusability and maintainability
Contract Monitor pattern used to verify contracts and authorize constituent systems services.	Our model in Figure 48 shows that Contract Monitor pattern is implemented in the HSH-SoS. It also implements both security tactics *"Authentication"* and *"Authorization"*, which help satisfying the security requirement.
The ***Pace-Layer*** used to establish well-defined interfaces of constituent systems	Our model in Figure 48 shows that the Pace-Layer pattern is implemented in the HSH-SoS and it supports the interoperability (establish well-defined interfaces of constituent systems at lower layers).
The ***Enterprise Service Bus*** pattern used for routing and transportation of services messages and events	Our model in Figure 48 shows that the ***Enterprise Service Bus*** pattern supports the connectivity and the data and protocol transformation, which helps the interoperability of the whole architecture
The ***SOA*** pattern used in the HSH-SoS architecture	Our model in Figure 48 shows that the ***SOA*** pattern is implemented in the HSH-SoS architecture and it supports interoperability, security, and reusability.

8.4.3. Build-Automation Systems

In [120], Gradle and Maven were compared based on *Performance, Flexibility,* and *User Experience* requirements. I only focused on *Performance*. The comparisons of Gradle and Maven on the performance in [120] is based on an empirical experiment. The authors ran performance measurements on a variety of projects under similar scenarios. To measure the impact on a typical library project, they converted the Apache Commons Lang 3 project

[133] from Maven to Gradle using a Java Library plugin [134]. Figure 60 shows the comparison results.

My results, shown in Section 7.3.5.2, are compatible with the comparison results in [120]. The experiment reported in [120] shows that Gradle is better than Maven for the performance. This is explained by performance support features in Gradle including parallelism (Gradle allows parallel execution of tasks and intra-task work through a Worker API) and the incremental build of subtasks (when Gradle discovers that the input or output of a task has changed between build runs, the task is executed again). As shown in Figures 56 and 57, Gradle, in fact, has a higher satisfaction level for the *Performance*, which is **(90)**. This is because of the implementation of the three performance tactics *"Introduce Concurrency", "Resource Pooling",* and *"Resource Scheduling"*. The *"Introduce Concurrency"* tactic allows parallel execution of tasks such that the blocked time can be reduced. The *"Resource Pooling"* tactic enables to share threads, database connections, and sockets. While the *"Resource Scheduling"* tactic allows to schedule processors, buffers, and networks.

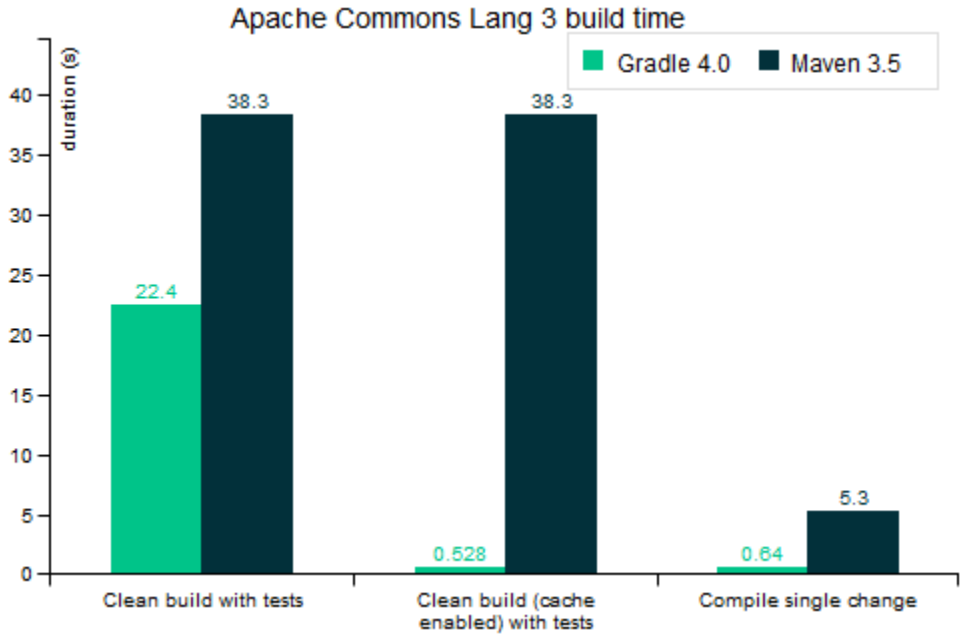


Figure 60 Comparison results between Gradle and Maven in terms of performance. Adapted from [120]

Table 54 summarizes the comparisons' results for this case study.

Table 54 Comparisons with [120]'s results

[120]'s Results	Our Results
Gradle is faster than Maven	According to Figures 56 and 57, Gradle is, in fact, faster than Maven and has a higher satisfaction level for the Performance, which is **(90)**. This is because of the implementation of the three performance tactics *"Introduce Concurrency," "Resource Pooling," and the "Resource Scheduling".* The tactic *"Introduce Concurrency"* allows parallel execution of tasks such that the blocked time can be reduced. The tactic "*Resource Pooling"* enables to share threads, database connections, and sockets. While the tactic *"Resource Scheduling"* allows to schedule processors, buffers, and networks, resulting in increasing the satisfaction level of the Performance for Gradle compared with Maven

8.5. Answers to Research Questions

In this section, I analyze our approach in light of the research questions presented in Chapter 1.

RQ1: Can I connect a software implementation to quality attributes in an objective (concrete) way to be able to evaluate the software architecture?

In this thesis, I presented an approach to evaluate software architectures that helps architects assess the potential of a proposed/chosen architecture to deliver a system capable of fulfilling required quality requirements and to identify any potential risks. The approach connects software implementation to quality attributes using the implemented patterns/tactics, in a software, combined with other criteria such as the importance values of the NFRs. I used *Archie* to extract the patterns and tactics from the source code of the

software architectures. A model of the software architectures in terms of their implemented patterns and tactics was created to support the evaluation of the architectures.

The results of applying our approach to the case studies and the comparisons with benchmark results from the literature show that the implemented patterns and tactics can be used to connect software implementation to quality attributes for an objective evaluation of software architectures. The results confirm that architectural patterns and tactics are reusable building blocks for software development, and they affect (positively and negatively) various quality attributes. For example, in the cyber fusion center case study, the results of Inoubli et al. [114] showed that Storm and Flink use the *"Checkpoint"* tactic for fault tolerance, while Spark uses recovery techniques. Our results in Section 7.1.5.3 show that both Storm and Flink implement the *"Checkpoint"* tactic to record consistent states and have a path to roll back to them if necessary, while Spark uses the *"Active Redundancy"* tactic for recovery, preparation, and repair of errors. The observation that Spark's response time is better than Flink's and Storm's has been addressed in [114]. This is also shown in our approach by a high satisfaction level of Performance for Spark, which is **(76)** and it is **(43)** for both Storm and Flink, as shown in Figure 42 in Chapter 7. The high satisfaction level for Spark came from applying the ***Shared-Repository*** pattern, which speeds up the data access in Spark. This helped satisfying performance in Spark more than in Flink and Storm. This observation is also has been addressed in [114], [127], [128], and [129].

Another example is in the Gradle and Maven case study, the authors in [120] showed that Gradle is faster than Maven. This has been shown in our results in Chapter 7. The fact that Gradle is faster than Maven comes from implementing the three performance tactics *"Introduce Concurrency", "Resource Pooling",* and *"Resource Scheduling"* in Gradle, which eventually improves the *Performance* of Gradle.

Therefore, knowing the characteristics, strengths, and weaknesses of each architecture pattern and tactic is necessary in order to determine whether they promote or impede the satisfaction of quality requirements in a system. This would lead us to answer yes to the primary question of this thesis. I can connect software implementation to quality attributes using their implemented patterns and tactics considering the importance values of given NFRs, and this can enable a more informed evaluation by architects.

RQ2: Is it possible to determine (extract) the patterns/tactics used by a software architecture?

Although a manual search of the implemented patterns and tactics in an implementation is possible, it is not practical for large systems. Different alternative methods have been used in the literature, such as:

a. *Archie* [11][12][13] (a tool that relies primarily on information retrieval and machine learning techniques for discovering tactics in code).

b. Matching methods between the provided services of a framework and its patterns/tactics [16].

c. *Pattern instantiation* (assigning the roles defined in a pattern to concrete classes, responsibilities, methods, and attributes of a practical design) [70].

d. Matching method between the problem statement of architecture and the applied patterns [18].

e. *Matrix-based approach* [25].

f. *Graph isomorphism-based approach* [26]. The system design is represented in one graph and the design patterns, to be recovered, in another graph.

g. Various tools, such as *Design Pattern Miner (DP Miner)* [25], *Architecture Reconstruction Plugin for Eclipse (MARPLE)-DPD* [29], *Fujaba* [33], *DP++* [34], *SPOOL* [35], *Osprey* [36].

h. *Logic inference* (SOUL [37]) based on inter-class-based code idioms and naming conventions.

i. The integration of existing tools – *Columbus* [40] and *Maisa* [41].

j. *Multilayered semiautomatic approach* [47].

k. Determination of the tactics from the solution and quality consequence sections of a pattern's documentation [69].

I can, therefore, say yes, patterns and tactics can be extracted either *manually, semi-automatically,* or *fully-automatically* from software source code.

In this thesis, I used *Archie* [11][12][13] to extract the patterns and tactics from software implementation. According to Mirakhorli's work [12], *Archie* is an extensible tool such that additional tactics can be added by defining terms related to tactics, with their weights. Because our goal is to extract both patterns and tactics, I extended *Archie* to detect both patterns and tactics by adding additional terms for patterns and tactics. *Archie* also allows finding where these tactics/patterns are implemented. It has an interactive interface so I can run more than one architecture, and it is a plugin of the Eclipse platform. Archie makes the extraction process faster by decreasing the time and effort spent searching for the patterns/tactics in the code.

RQ3: Is it possible to evaluate software architectures based on their implemented patterns and tactics?

To answer this research question, I created a model of the evaluated software architectures in terms of their implemented patterns and tactics. The model supports the evaluation of the quality attributes by providing the contribution values of the patterns on the quality attributes. This would provide a rationale about which quality attributes are improved and which are not by which pattern(s)/tactics(s). Knowing the characteristics, strengths, and weaknesses of each architecture pattern/tactic are necessary to determine whether they push or pull a system toward or away from a quality requirement.

Providing a rationale about software architectures in terms of their implemented patterns and tactics that considers the importance values of NFRs and the tradeoffs between the requirements and shows which quality attributes are either improved or decreased by which patterns/tactics, helps evaluate software architectures. Such a rationale could enable a more informed assessment by architects. I can therefore answer yes to this research question.

However, evaluating the candidate software architectures only based on their implemented patterns and tactics considering the importance values of NFRs may not be sufficient. Architects still need to consider other criteria such as stability, maturity, and

community of a framework. As mentioned previously, I plan to consider this in our future work.

8.6. Chapter Summary

This Chapter summarizes the evaluation and validation process of our proposed approach in this thesis. The following list summarizes the evaluation methods I used to evaluate and validate our approach:

- *Literature Review*: I compared the approach with related approaches based on several criteria.
- *Real-World Case Studies*: I validated the approach by applying it to real-life case studies and illustrate that it can be applied in different contexts. I analyzed the results to be able to answer our research questions. The details on these case studies are available in Chapter 7. A summary of what I learned in these studies is presented here.
- *Manual Checking in Source Code*: I manually checked the source code for the occurrences of patterns and tactics found by *Archie*.
- *Comparisons with Benchmark Results from the Literature*: I compared the results of the application of our approach on real-world case studies to benchmark comparisons from the literature.

Chapter 9. Conclusions and Future Work

The final chapter of this thesis summarizes its contributions and future work items.

9.1. Contributions

The contributions of the thesis are characterized as major, and minor. They are captured respectively in Table 55 and Table 56.

Table 55 Major thesis contributions

Deliverable	**Value**
Software architecture evaluation approach that helps assess an implementation based on the uncovering, analysis and modeling of the architectural patterns and tactics used in the code. See Chapters 4, 5, and 6.	The evaluation is supported by a provided rationale about the evaluated software architectures in terms of their implemented patterns and tactics. It considers the importance values of NFRs. Such a rational allows architects to choose among different architectures, analyze and understand an architecture, and compare several software tools.
An inventory of reusable models for tactics and patterns, resulting in the creation of a model for the software architectures. See Chapter 5.	It allows architects to use the models in the inventory to model their own architectures or systems. They can use these models as is or customize them according to their architecture needs. Allows researchers to • Use these models to model other architectures/systems in different contexts.

Table 56 Minor thesis contributions

Deliverable	Value
Adapted *Archie* with our added tactics and patterns. This contribution has been made by performing a literature review to find common architectural patterns and tactics. The results of *Archie* have been validated by manually searching for the occurrences of the patterns and tactics detected by *Archie* in the source code of the software architectures. See Chapter 4	This is an alternative to the training approach. Allows architects to use *Archie* with our added patterns and tactics as is or improve it by adding more patterns and tactics to *Archie*.
Review of the existing software architectures evaluation approaches. See Chapter 2	Allows architects to evaluate our approach. Allows other researchers to improve the approach and investigate it in a different context.
Review of the existing techniques for modelling a software architecture. See Chapter 2	Allows other researchers to know those techniques, compare between them, and choose the best one.
Review of the existing techniques for connecting software implementation to quality attributes. See Chapter 2	Allows other researchers to know those techniques, compare between them, and choose the best one.
Review of the existing techniques for determining patterns and tactics in an implementation. See Chapter 2	Allows other researchers to know those techniques, compare between them, and choose the best one.
Performing literatures on the relevant patterns, tactics, and NFRs of a system in a specific context. See Chapter 4 and 6.	Allows other researchers to use these patterns and tactics as is in their own research or update them by finding other patterns and tactics in different contexts or the same contexts used in this thesis.

9.2. Threats to Validity

Threats to validity include construct, internal, and external validity. I discuss the threats which potentially impacted our work and how I attempted to mitigate them.

External Validity evaluates the generalizability of the approach. One important external threat is that only three contexts were covered in this thesis, and further case studies from these contexts and others would be beneficial. Another external threat is that I target one modeling language (GRL) to model software architectures and one tool (*Archie*) to determine patterns and tactics of source codes. Other modeling languages and tools may not provide the same results as the one presented here.

Construct Validity aims to assess to which extent the case studies and the benchmark comparisons answer our research hypothesis. An important threat is the limitation of the benchmark comparisons and the inferred quality attributes from the literature. Additionally, the usability of the approach based on the experience of others was not assessed.

Internal Validity reflects the extent to which a work minimizes bias so that a causal conclusion can be drawn. A threat to validity is that the search for specific patterns or tactics was solely performed by the thesis author. Similarly, the literature done by the thesis author under the supervision of another. Another important threat is the accuracy of the contribution values of pattern/tactics to design decision affecting the NFRs. These values have been determined based on my analysis of the literature. Others may come to different values which could affect the evaluation results.

9.3. Limitations to the Approach

I identified various limitations to our work. Addressing these limitations in future work have the potential to improve the approach.

- My approach is based on the presence of architectural tactics and patterns in code. It is however possible that patterns/tactics could be implemented the wrong way and not provide their expected benefits. Although our initial validation with case

studies such as Gradle and Maven provide some confirmation to our assumption, more case studies will however be required.

- The task of identifying the related terms was conducted manually, by looking at different descriptions of a pattern/tactic from different sources. I then performed a validation by checking the existence of the determined terms in the source code of the software architectures. The manual identification of the related terms in different descriptions from different sources and the existence of the related terms written by developers in the source code gave us confidence that each of the identified terms was indeed representative of its relevant pattern/tactic. However, this is an obstacle to the general applicability of the approach to domains not considered in this thesis. Moreover, it is difficult to ensure that all related terms of a given pattern or tactic have been identified. For example, there could be terms that I failed to find.
- The task of calculating the contribution values was conducted manually by matching the qualitative contributions from the literature to our assumed GRL contribution values. Our assumed values are based on our understanding of the contribution values of the patterns and tactics in the literature. However, it is difficult to ensure that all the assumed contribution values are 100% accurate, especially the ones in the range between 1 to 99 and -1 to -99.
- I do not check the whole code of the software architectures to identify architectural patterns and tactics unreported by *Archie* (False Negatives). Consequently, I do not calculate the Recall value. This need very deep knowledge of the source code particularly given that many architectural patterns can have several forms of implementation.
- The determination of patterns and tactics in the source code of a software architecture is semi-automatic (using both *Archie* and the manual search). There is no tool that can detect both patterns and tactics from source code. Archie was designed for the detection of tactics. Unlike a tactic, a pattern usually consists of multiple elements forming its structure, of relationships between these elements and of a described behavior of the elements. I added the capability to identify architectural patterns. However, manual work is needed to confirm the effective

presence of reported candidate patterns in the code. Therefore, a completely automated way to detect patterns/tactics in a source code is needed.

- The NFRs derived from patterns/tactics such as performance might not be sufficient to be able to evaluate software architectures. I consider the result provided by our approach as one component of the criteria for a final decision on evaluating a software architecture. Other criteria including the cost, stability, maturity, community support might also be considered.

- The limitation of the inferred quality attributes (i.e. *performance, availability,* and *reliability*) which have been compared to benchmark results from the literature. There might be other benchmark comparisons not considered.

- I do not consider the possible interactions between the different patterns and tactics in an implementation. These interactions could affect NFRs in a negative or positive way.

9.4. Future Work

The research that was done in this thesis provides enough value to be used as-is. However, there are some areas of the research that can be further enhanced to mitigate the threats and limitations identified in the previous sections. Besides, this research opens up some new research directions and opportunities. The following subsections describe some potential future research areas and enhancements.

- **Using Indicators in the GRL model**

  In this thesis, I calculated the contributions values manually based on our matching of the contributions of the patterns and tactics to the NFRs from the literature and the assumed GRL contributions values. To get those contributions values automatically and based on real data, I can use indicators (one of the GRL notations) as an input to our model. This would allow to get real data as input so that the contribution values of a pattern/tactic to a given NFR can be automatically calculated.

- **Extending the approach to find anti-patterns and tactical vulnerabilities**

  Tactical vulnerabilities and anti-patterns concepts are incorrect implementations of tactics or patterns, which cause software bugs and vulnerabilities in the architecture of a system [135]. Santos et al. [135] referred to the vulnerabilities related to security tactics. They investigated the most common types of vulnerabilities associated with security tactics and tried to fix them to ensure appropriate implementations of security related-design decisions in code. Our approach can be extended to consider anti-patterns and tactical vulnerabilities.

- **Considering other evaluation criteria**

  Evaluating the candidate software architectures based on their implemented patterns and tactics may not be sufficient to make an assessment particularly when choosing among different implementations. In the future, I might integrate other criteria such as maturity, stability, or the community of a software architecture.

- **Calculating the number of the False Negatives in the source code of a software architecture**

  Calculating the number of the FN and consequently the values of recall would be one of our future work. This can be performed by hunting for the occurrences of the detected patterns and tactics manually in the whole code of a software architecture or using another tool as a basis.

- **Extraction based on multiple-words terms associated to patterns and tactics**

  In this thesis, I only determine single word terms related to a pattern/tactic. This is in accordance with Mirakhorli approach for the definition of terms related to tactics in Archie. It would be interesting to consider multiple words terms.

- **Application to more case studies for validation**

  In this thesis, I validated the approach by applying it to different case studies from different contexts. I also assessed its compatibility with the documentation of the real examples by comparing the inferred quality attributes with benchmark comparison results from the literature. Furthermore, I validated the approach by comparing it to existing evaluation approaches. In the future, I can validate the

approach using empirical validation. This can be done by having architects apply the approach and getting their feedback about its applicability and usefulness in practice.

- **Exploring additional sources to find patterns and tactics**

  In this thesis, only the source code was used to find patterns and tactics in a software architecture implementation. Additional sources such as the software documentation or direct information from developers can be explored.

References

[1] Cervantes, H., Elizondo, P.V., and Kazman, R.: "A principled way to use frameworks in architecture design", *IEEE Software*, March/April, 46-53, (2013).

[2] Grau, G., and Franch, X.: "A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures", *European Conference on Software Architecture (ECSA)*, pp 139-155, (2007).

[3] Zalewski, A.: "Modeling and Evaluation of Software Architecture", *Warsaw University of Technology Publishing Office*, (2013).

[4] Kazman, R., Bass, L., Abowd, G., and Webb, M.: "SAAM: A Method for Analyzing the Properties of Software Architectures", *Proc. 16th International Conference of Software Engineering*, pp. 81-90, (1994).

[5] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, S. J.: "The Architecture Tradeoff Analysis Method", *Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 68-78, (1998).

[6] Luckham, D., John, K., Augustin, L., Vera, J., Bryan, D., and Mann, W.: "Specification and Analysis of System Architecture using RAPIDE", *IEEE Transactions on Software Engineering*, 21(4):336-335, (1995).

[7] Aquilani, F., Balsamo, S., and Inverardi, P.: "Performance Analysis at the Software Architectural Design Level", *Performance Evaluation*, vol. 45, pp. 147-178, (2001).

[8] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513803, (last accessed 2020/02/10).

[9] Henver, A.R., March, A.T., Park, J., and Ram, S.: "Design science in information systems research", *MIS Quarterly*, Vol. 28, No.1, pp. 75–105, (March 2004).

[10] Hevner, A.: "Three Cycle View of Design Science Research", *In Scandinavian Journal of Information Systems*, (January 2007).

[11] Mirakhorli, M.: "Preserving the Quality of Architectural Tactics in Source Code", *The Institutional Repository at DePaul University*, College of Computing and Digital Media, (2014).

[12] Mirakhorli, M., and Cleland-Huang, J.: "Detecting, Tracing, and Monitoring Architectural Tactics in Code", *IEEE Transactions on Software Engineering*, Volume: 42, Issue 3, pp 205-220, (2016).

[13] Mirakhorli, M., Fakhry, A., Grecho, A., Wieloch, M., and Cleland-Huang, J.: "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, (2014).

[14] Mussbacher, G., Weiss, M., and Amyot, D.: "Formalizing Architectural Patterns with the Goal-oriented Requirement Language", *Proceedings of the Fifth Nordic Conference on Pattern Languages of Programs*, (2007).

[15] Tvedt, R.Z., Lindvall, M., and Costa, P.: "A Process for Software Architecture Evaluation using Metrics", *Proceedings of the 27 th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW-27'02)*, (2003).

[16] Johnson, R.E.: "How frameworks compare to other object-oriented reuse techniques", *Communications of the ACM*, 40(10), 39-42, (1997).

[17] Meusel, M., Czarnecki, K., and Köpf, W.: "A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext", *European Conference on Object-Oriented Programming ECOOP*, pp 469-510, (1997).

[18] Beck, K., and Johnson, R., "Patterns Generate Architecture", *ECOOP '94 Proceedings of the 8th European Conference on Object-Oriented Programming*, pp 139-149, London, UK, (1994).

[19] Rasool, G. and Mäder. P.: "Flexible design pattern detection based on feature types", In *ASE*, pp, 243–252, (2011).

[20] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G.: "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley Professional*, First edition, 416 pages, (Oct. 31, 1994).

[21] Mirakhorli, M., Mäder, P., and Cleland-Huang, J.: "Variability points and design pattern usage in architectural tactics", *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 52, pp 1-11, (2012).

[22] James R. Groff, Paul N. Weinberg, and Oppel, A.: "SQL The Complete Reference", *McGraw-Hill Education*, Third Edition, (Sept. 2, 2009).

[23] Shi, N., and Olsson, R.A.: "Reverse Engineering of Design Patterns from Java Source Code", *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp 123-134, https://doi.org/10.1109/ASE.2006.57, (September 2006).

[24] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S.T.: "Design pattern detection using similarity scoring", *IEEE TSE*, vol. 32, no. 11, pp. 896–909, (2006).

[25] Dong, J., Lad, D.S., and Zhao, Y.: "DP-miner: Design pattern discovery using matrix," *in ECBS'07*, pp. 371–380, (2007).

[26] Yu, D., Ge, J., and Wu, W.: "Detection of Design Pattern Instances Based on Graph Isomorphism", *IEEE 4th International Conference on Software Engineering and Service Science*, (May 2013).

[27] Dong, J., Sun, Y., and Zhao, Y.: "Design Pattern Detection by Template Matching", *Proceedings of the 2008 ACM symposium on Applied computing*, pp 765-769, https://doi.org/10.1145/1363686.1363864, (March 2008).

[28] Gupta, M., Pande, A., Rao, R.S., and Tripathi, A.K.: "Design Pattern Detection by Normalized Cross Correlation", *International Conference on Methods and Models in Computer Science (ICM2CS)*, New Delhi, India, (December 2010).

[29] Zanoni, M., Fontana, F.A., and Stella, F.: "On applying machine learning techniques for design pattern detection", *The Journal of Systems and Software*, pp 102-117, (2015).

[30] Dwivedi1, A.K., Tirkey, A., Ray, R.B., and Rath, S.K., "Software Design Pattern Recognition using Machine Learning Techniques", *7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Noida, India, (2018).

[31] Uchiyama, S., and Kubo, A.: "Design Pattern Detection using Software Metrics and Machine Learning", *Fifth International Workshop on Software Quality and Maintainability (SQM)*, (2011).

[32] Arcelli, F., and Cristina, L.: "Enhancing Software Evolution through Design Pattern Detection", *Third IEEE Workshop on Software Evolvability*, (2007).

[33] Niere, J., Shafer, W., Wadsack, J.P., Wendehals, L., and Welsh, J.: "Towards pattern-based design recovery", *In International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, pp338–348, (May 2002).

[34] Bansiya, J.: "Automating design-pattern identification –DP++ is a tool for C++ programs", *Dr. Dobbs Journal*, (1998).

[35] Keller, R., Shauer, R., Robitaille, S., and Pag´e, P.: "Pattern based reverse-engineering of design components", *In Proc. Of the 21st International Conference on Software Engineering*, IEEE Computer Society Press, pp 226 235, (May 1999).

[36] Asencio, A., Cardman, S., Harris, D., and Laderman, E.: "Relating expectations to automatically recovered design patterns", *In WCRE*, pp 87–96, (2002).

[37] Fabry, J., and T. Mens, T.: "Language Independent Detection of Object-Oriented Design Patterns", *Computer Languages, Systems and Structures*, (February 2004).

[38] Gueh´eneuc, Y.G., Shraoui, H., and Zaidi, F.: "Fingerprinting design patterns", *In Proceedings of the 11th Working Conference on Reverse Engineering*, pp 172–181, (Nov 2004).

[39] Ferenc, R., Gustafsson, J., M¨uller, L., and Paakki, J.: "Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa", *Acta Cybernetica*, 15:669–682, (2002).

[40] Ferenc, R., Besze´des, A., Tarkiainen, M., and T. Gyim´othy, T.: "Columbus – Reverse Engineering Tool and Schema for C++", *In Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society, pp 172–181., (Oct. 2002).

[41] Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., and Verkamo, A.: "Software metrics by architectural pattern mining", *In Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pp 325–332, (2000).

[42] Ferenc, R., Gustafsson, J., M¨uller, L., and Paakki, J.: "Recognizing design patterns in C++ programs with the integration of Columbus and Maisa", *Acta Cybern*., 15(4):669–682, (2002).

[43] Wendehals, L.: "Specifying patterns for dynamic pattern instance recognition with uml 2.0 sequence diagrams", *In Proceedings of the 6th Workshop Software Reengineering (WSR2004)*, pp 63–64, (May 2004).

[44] Balanyi, Z., and Ferenc, R.: "Mining design patterns from C++ source code", *In Proc. of the International Conference on Software Maintenance*, IEEE Computer Society Press, pp 305–314, (September 2003).

[45] Antoniol, G., Fiutem, R., and L. Cristoforetti, L.: "Design pattern recovery in object-oriented software", *In Proc. of the 6th International Workshop on Program Comprehension,* IEEE Computer Society Press, pp 153–160, (June 1998).

[46] Ferenc, R., Besz´edes, A., Fulop, L., and Lele, J.: "Design pattern mining enhanced by machine learning", *In ICSM*, pp 295–304, (2005).

[47] Guéhéneuc, Y.G., and Antoniol, G.: "DeMIMA: A multilayered approach for design pattern identification", *IEEE TSE*, vol. 34, no. 5, pp. 667–684, (2008).

[48] Zhu, L., Babar, M.A., and Jeffery, R.: "Mining patterns to support software architecture evaluation", *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, (June 2004).

[49] Marinescu, F.: "EJB design patterns: advanced patterns, processes, and idioms", New York: *John Wiley*, (2002).

[50] Ryoo, J., Laplante, P., and Kazman, R.: "Revising a Security Tactics Hierarchy through Decomposition, Reclassification, and Derivation", *IEEE Sixth International Conference on Software Security and Reliability Companion*, (2012).

[51] Yu, E., "Modelling Strategic Relationships for Process Reengineering", PhD. Thesis, *University of Toronto*, (1995).

[52] Kassab, M., and El-Boussaidi, G.: "Towards Quantifying Quality, Tactics and Architectural Patterns Relations", The *25th International Conference on Software Engineering and Knowledge Engineering*, (June 2013).

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

[54] 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/).

[55] The Open Group: ArchiMate® 2.0 Specifi cation. 2009-2012. The Open Group. Available online. *The Open Group*, (2012).

[56] Tyree, J., and Akerman, A., "Architecture Decisions: Demystifying Architecture", *IEEE Software*, pp. 19-27, (Mar.-Apr. 2005).

[57] Zimmermann, O., Zdun, U., Gschwind, T., and Leymann, F.: "Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method", *WICSA*, pp. 157-166, 18-21, (Feb 2008).

[58] Araujo, I., and Weiss, M.: "Linking Patterns and Non-Functional Requirements", *Conference on Pattern Languages of Programs (PLoP)*, (2002).

[59] Mohsin, A., Janjua, N. K., Islam, M. S., and Neto, G. V. V.: "A Taxonomy of Modeling Approaches for Systems-of-Systems Dynamic Architectures: Overview and Prospects", *IEEE SoSE 2019 conference*. (23 May 2019).

[60] Avritzer, A. and Weyuker E. J.: "Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews", *Empirical Software Engineering*, 4(3), pp 199-215, (1999).

[61] Lindvall, M., Tvedt, R. T., and Costa, P., "An empirically based process for software architecture evaluation", *Empirical Software Engineering*, 8(1):83-108, 2003.

[62] Klein, M. and Kazman, R., "Attribute-Based Architectural Styles", *CMU/SEI-99-TR-22, Software Engineering Institute*, Carnegie Mellon University, 1999.

[63] Vieira, M. E. R., Dias, M. S., and Richardson, D. J.: "Analyzing software architectures with Argus-I", *Proc. 22nd International Conference on Software Engineering*, pp. 758- 761, (2000).

[64] Wang, J., He, X., and Deng, Y.: "Introducing Software Architecture Specification and Analysis in SAM Through an Example", *Information and Software Technology*, 41(7):451- 467, (May 1999).

[65] Smith, C. and Williams, L.: "Performance Solutions", ISBN 0- 201- 72229-1, *Addison-Wesley*, (2002).

[66] "CBAM: Cost Benefit Analysis Method http://www.sei.cmu.edu/ata/products_services/cbam.html, (Last accessed 2020/01/04).

[67] Lassing, Nico, Ph.D. Thesis, "Architecture-Level Modifiability Analysis", Ph.D. thesis, *Free University Amsterdam*, (February 2002).

[68] Thomas J.D.: "Architecture Assessment of Information-System Families", Ph.D. Thesis, *Department of Technology Management, Eindhoven University of Technology*, (February 2002).

[69] Zhu, L.: "Software Architecture Evaluation for Framework-based Systems", Ph.D. thesis, *University of New South Wales*, (2006).

[70] Aguiar, A., and David, G.: "Patterns for effectively documenting frameworks", *Transactions on Pattern Languages of Programming II*, 79-124, Springer, (2011).

[71] Heesch, U.V., Avgeriou, P., and Hilliard, R.: "A documentation framework for architecture decisions", The *Journal of Systems and Software 85*, pp795– 820, (2012).

[72] Sena, B., Allian, A.P., and Nakagawa, E.Y.: "Characterizing Big Data Software Architectures: A Systematic Mapping Study", In *Proceedings of SBCARS 2017*, Fortaleza, CE, Brazil, September 18–19, (2017).

[73] Sena, B., Garces, L., Allian, A.P., and Nakagawa, E.Y.: "Investigating the Applicability of Architectural Patterns in Big data Systems", *Pattern Languages of Programs (PLoP)*, Portland, Oregon, USA, (OCTOBER 24–26 2018).

[74] Ryoo, J., Kazman, R., and Anand, P.: "Architectural Analysis for Security", *IEEE Security & Privacy*, Vol 13, Issue 6, (Nov.-Dec. 2015).

[75] Bass, L., Clements, P., and Kazman, R.: "Software Architecture in Practice", *Addison-Wesley,* (2012).

[76] Carey, J. and Carlson, B.: "Framework Process Patterns: Lessons Learned Developing Application Frameworks", *Addison-Wesley*, (2002).

[77] Kazman, R. and Cervantes, H.: "Designing Software Architectures: A Practical Approach", *Addison-Wesley Professional,* 1st edition. (May 2016).

[78] Lattanze, A. J: "The Architecture Centric Development Method", *Carnegie Mellon University Report-Institute for Software Research International School of Computer Science CMU-ISRI-05-103*, (2005).

[79] Apache Storm, Yu, F.: https://storm.apache.org, (last accessed 2019/09/2).

[80] Apache Metron, Apache Metron, metron.apache.org, (last accessed 2019/09/2).

[81] Apache Flink, flink.apache.org, (last accessed 2019/09/2).

[82] Apache Spark, https://spark.apache.org, (last accessed 2019/09/5).

[83] Angular, https://angular.io/, (last accessed 2019/10/25).

[84] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: "Pattern-Oriented Software Architecture: A System of Patterns", vol. 1, *John Wiley & Sons*, (1996).

[85] Buschmann, F., Henney, K., and Schmidt, D.C.: "Pattern-Oriented Software Architecture", vol. 4, *Wiley*, (2007).

[86] Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., and Merlo, E.: "Recovering traceability links between code and documentation", vol. 28, *IEEE Trans. Softw. Eng*, p. 970–983, (2002).

[87] Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., and Romanova, E.: "Best practices for automated traceability Computer", vol. 40, p. 27–35, (2007).

[88] Biggerstaff, T.: "Design Recovery for Maintenance and Reuse", *IEEE Computer*, (July 1989).

[89] Biggerstaff, T., Mitbander, B., and Webster, D.: "The Concept Assignment Problem in Program Understanding", *Proc. Int'l Conf. Software Engineering*, pp. 482–498, (May 1993).

[90] Amyot, D., Ghanavati, S., Horkoff, J., Mussbacher, G., Peyton, L., and Yu, E.: "Evaluating Goal Models within the Goal-oriented Requirement Language", *International Journal of Intelligent Systems*, pp 841-877, (August 2010).

[91] Chung, L., Nixon, B., and Yu, E.: "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design", *Proc. 1st Int. Workshop on Architectures for Software System*, pp 31-43, (1994).

[92] Chung, L., Cooper, K., and Yi, A.: "Developing adaptable software architectures using design patterns: an NFR approach", *Computer Standards & Interfaces*, vol. 25, pp. 253–260, (2003).

[93] Mehta, R., Ruiz-López, T., Chung, L., and Noguera, M.: "Selecting among Alternatives using Dependencies: An NFR approach", *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, New York, NY, USA, pp 1292-1297, (2013).

[94] Munro, S., Liaskos, S., Aranda, J.: "The Mysteries of Goal Decomposition", In: iStar 2011. *CEUR-WS*, vol. 766, pp. 49–54, (2011).

[95] Mussbacher, G., Amyot, D., Heymans, P.: "Eight Deadly Sins of GRL", *In: iStar CEUR-WS*, vol. 766, pp. 2–7, (2011).

[96] Maté, A., Trujillo, J., Franch, X.: "Adding semantic modules to improve goal-oriented analysis of data warehouses using I-star", *JSS 88*, pp 102–111, (2014).

[97] Moody, D.L., Heymans, P., Matulevičius, R.: "Visual syntax does matter improving the cognitive effectiveness of the i* visual notation", Requirement. Engineering. 15(2), pp 141–175, (2010).

[98] Ong, H., Weiss, M., and Araujo, I.: "Rewriting a Pattern Language to Make it More Expressive", *Proceedings of ChiliPLoP*, (2003).

[99] https://en.wikipedia.org/wiki/Analytic_hierarchy_process, (last accessed 2019/12/23).

[100] Akhigbe, O. S.: "Business Intelligence-Enabled Adaptive Enterprise Architecture", Master Thesis, (2014).

[101] Akhigbe, O., Alhaj, M., Amyot, D., Badreddin, O., Braun, E., Cartwright, N., Richards, G., and Mussbacher, G.: "Creating Quantitative Goal Models: Governmental Experience", *International Conference on Conceptual Modeling*, Lecture Notes in Computer Science book series (LNCS), volume 8824, pp 466-473, (2014).

[102] Saaty, T. L.:"What is the Analytic Hierarchy Process?", *Mathematical Models for Decision Support*, Section 3, Springer Berlin Heidelberg, pp 109-121, (1998).

[103] https://en.wikipedia.org/wiki/Group_decision-making, (last accessed 2019/12/23).

[104] Milhem, H., Weiss, M., and Somé, S.: "Extraction of Architectural Patterns from Frameworks and Modeling their Contributions to Qualities", In*: 26th Proc of Conf. on Pattern Languages of Programs (PLoP).* Ottawa, (October 2019).

[105] Milhem, H., Weiss, M., and Somé, S.: "Modeling and Selecting Frameworks in terms of Patterns, Tactics, and System Qualities". In *32th Proc of International Conference on Software Engineering and Knowledge Engineering (SEKE).* USA, (July 2020).

[106] Garces, L., Sena, B., and Nakagawa. E. Y.: "Toward an architectural patterns languages for Systems-of Systems", *HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. (PLoP)*, Canada-Ottawa, 24 pages, (October 2019).

[107] Harrison, N., "Improving Quality Attributes of Software Systems Through Software Architecture Patterns", Thesis, *University of Groningen, the Netherlands*, (2011).

[108] Harrison, N., and Avgeriou, P.: "Leveraging architecture patterns to satisfy quality attributes", In *proc. First European Conference on Software Architecture*, Madrid, Springer LNCS, (Sept 24-26, 2007).

[109] Bode, S., and Riebisch, M.: "Impact Evaluation for Quality-Oriented Architectural Decisions Regarding Evolvability", *European Conference on Software Architecture ECSA*, (2010).

[110] Alebrahim, A., Fassbender, S., Filipczyk, M., Goedicke, M., and Heisel, M.: "Towards a Reliable Mapping between Performance and Security Tactics, and Architectural Patterns", *EuroPLoP '15 Proceedings of the 20th European Conference on Pattern Languages of Programs*, Article No. 39, 43 pages, (2015).

[111] Me, G., Calero, C., and Lago, P.: "Architectural patterns and quality attributes interaction", *Qualitative Reasoning about Software Architectures (QRASA)*, (2016).

[112] Harrison, N., and Avgeriou, P.: "Implementing Reliability: The Interaction of Requirements, Tactics and Architecture Patterns", *Architecting Dependable Systems VII,* pp 97-122, (2010).

[113] Kassab, M., El-Boussaidi, G., and Mili, H.: "A Quantitative Evaluation of the Impact of Architectural Patterns on Quality Requirements", *Software Engineering Research Management and Applications,* pp 173-184, (2011).

[114] Inoubli, W., Aridhi, S., Mezni, H., Maddouri, M., and Nguifo, E. M.: "An experimental survey on big data frameworks", *Future Generation Computer Systems*, pp 546-564, 2018.

[115] Gradle User Manual, https://docs.gradle.org/current/userguide/userguide.html, (last accessed 2019/06/15).

[116] Apache Maven Project, https://maven.apache.org/what-is-maven.html, (last accessed 2019/08/20).

[117] Introduction to XML, https://www.w3schools.com/xml/xml_whatis.asp, (last accessed 2019/11/13).

[118] Domain - Specific language, https://en.wikipedia.org/wiki/Domain-specific_language, (last accessed 2019/11/13).

[119] Apache Groovy, https://en.wikipedia.org/wiki/Apache_Groovy, (last accessed 2019/11/13).

[120] Gradle vs Maven Comparison, https://gradle.org/maven-vs-gradle/, (last accessed 2019/11/2).

[121] Gradle vs. Maven, https://dzone.com/articles/gradle-vs-maven (Last accessed 2020/3/30).

[122] Gradle vs. Maven: Performance, Compatibility, Builds, & More, https://stackify.com/gradle-vs-maven/ (Last accessed 2020/3/30).

[123] Ant vs Maven vs Gradle, https://www.baeldung.com/ant-maven-gradle (Last accessed 2020/3/30).

[124] Gupta, K., Gradle vs Maven 2019: What's The Difference, https://www.freelancinggig.com/blog/2019/01/16/gradle-vs-maven-2019-whats-the-difference/ (Last accessed 2020/3/30).

[125] Why Use Gradle Instead of Ant or Maven, https://stackoverflow.com/questions/1163173/why-use-gradle-instead-of-ant-or-maven (Last accessed 2020/3/30).

[126] Introduction to the POM, https://maven.apache.org/guides/introduction/introduction-to-the-pom.html, (last accessed 2019/12/5).

[127] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., and Peng B.J. et al.: "Benchmarking streaming computation engines: Storm, flink and spark streaming,", In *IEEE International Parallel and Distributed Processing Symposium Workshops. IEEE*, pp. 1789–1792, (2016).

[128] Lopez, M.A., Lobato, A.G.P., and Duarte, O.C.M.B.: "A Performance Comparison of Open-Source Stream Processing Platforms", *GLOBECOM*, pp. 1-6, (2016).

[129] Hesse, G., and Lorenz, M.: "Conceptual Survey on Data Stream Processing Systems", *IEEE 21st International Conference on Parallel and Distributed Systems*, http://dx.doi.org/10.1109/ICPADS.2015.106, (2015).

[130] Kaur, D., Chadha, R., and Verma, N.: "Comparison of Micro-Batch and Streaming Engine on Real Time Data", *International Journal of Engineering Sciences & Research Technology*, 6(4), pp 756-761 (April 2017).

[131] Bartolini, I., and Patella, M.: "Comparing Performances of Big Data Stream Processing Platforms with RAM3S", In *Proc. SEBD*, (2017).

[132] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., and Markl, V.: "Benchmarking Distributed Stream Data Processing Systems", *IEEE 34th International Conference on Data Engineering (ICDE)*, Paris, France, 10.1109/ICDE.2018.00169, (2018).

[133] Apache Commons, http://commons.apache.org/proper/commons-lang/, (last accessed 2020/01/20).

[134] The Java Library Plugin, https://docs.gradle.org/current/userguide/java_library_plugin.html, (last accessed 2020/01/22).

[135] Santos, J. C. S, Peruma, A., Mirakhorli, M., Galstery, M., Vidal, J. V., and Sejfia. A.: "Understanding Software Vulnerabilities Related to Architectural Security Tactics: Am Emprical Investigation of Chromium, PHP and Thunderbird", *IEEE International Conference on Software Architecture (ICSA)*, Gothenburg, Sweden, DOI: 10.1109/ICSA.2017.39. (3-7 April 2017).

[136] Santos, M.Y., Esa, J.O., Costa, C., Galvao, J., Andrade, C., Martinho, B., Lima, F.V., and Costa, E.: "A Big Data Analytics Architecture for Industry 4.0", *World Conference on Information Systems and Technologies*, pp 175-184, Volume 2, (2017).

[137] Moreno, J., Serrano, M.A., Medina, E.F., Fernandez, E.B.: "Toward a Security Reference Architecture for Big Data", *DOLAP*, (2018).

[138] Carrez, F., Elsaleh, T., Gomez, D., Sanchez, L., Lanza, J., and Grace, P.: "A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability", *European Conference on Networks and Communications (EuCNC)*, Finland, (2017).

[139] He, A., Shen, J., Wang, Y., and Liu, L.: "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*, (2018).

[140] Sang, G.M., Xu, L., and de Vrieze, P.: "Simplifying Big Data Analytics Systems with a Reference Architecture", *Working Conference on Virtual Enterprises: Collaboration in a Data-Rich World*, pp 242-249, (2017).

[141] Talon, B.: "Data-Centric Security Platform: Bringing Order to Data Security Chaos", http://bluetalon.com/data-centric_security/, (2016).

[142] SQRRL. "Big Data and Data Centric Security", http://sqrrl.com/_media/Data-Centric-Security-WP-final-.pdf, (2014).

[143] Kandogan, E., Soroker, D., Rohall, S.L., Bak, P., Ham, F.V., Lu, J., Ship, H., and Fu, C. Wang: "A reference web architecture and patterns for real-time visual analytics on large streaming data", *Proceedings of SPIE - The International Society for Optical Engineering*, (2013).

[144] Kim, C.H., Park, K., Fu, J., and Elmasri, R.: "Architectures for Streaming Data Processing in Sensor Networks", *The 3rd ACS/IEEE International Conference onComputer Systems and Applications*, Cairo, Egypt, (2005).

[145] Dhaouadi, J., and Aktas, M.: "On the Data Streaming Processing Frameworks: A Case Study", *3rd International Conference on Computer Science and Engineering (UBMK)*, (2018).

[146] Sang, G.M., Xu, L., and de Vrieze, P.: "A Reference Architecture for Big Data Systems", *10th International Conference on Software, Knowledge, Information Management & Applications (SKIMA)*, (2016).

[147] Singh, D., and Reddy, C.K.: "A survey on platforms for big data analytics", *Journal of Big Data*, (2015).

[148] Mahfuz, S., Isah, H., Zulkernine, F., and Nicholls, P.: "Detecting Irregular Patterns in IoT Streaming Data for Fall Detection", *IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, (2018).

[149] Marcu, O., Costan, A., Antoniu, G., and Pérez-Hernández, M.S.: "Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks", *The IEEE International Conference on Cluster Computing*, Taipei, Taiwan, (2016).

[150] Verbitskiy, I., Thamsen, L., and Kao, O.: "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), (2016).

[151] Cao, R., and Gao, J.: "Research on Reliability Evaluation of Big Data System", *IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, (2018).

[152] Ullah, F., and Babar, M.A: "Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review", *ArXiv*, (2018).

[153] Muller, H.A., Kienle, H.M., and Stege, U.: "Autonomic Computing Now You See It, Now You Don't. In Software Engineering", *Lecture Notes in Computer* Sciences, *Springer Berlin Heidelberg*, Vol. 5413, Berlin, pp 32–54, (2009).

[154] Barnes, J.M., Garlan, D., and Schmerl, B.: "Evolution styles: Foundations and models for software architecture evolution", *Software and Systems Modeling*, 649–678. DOI:https://doi.org/10.1007/s10270-012-0301-9. (2014).

[155] Kazman, R., Nielsen, C., and Schmid, K.: "Understanding Patterns for System-of-Systems Integration Understanding Patterns for System-of- Systems Integration", Technical Report CMU/SEI-2013-TR-017, *Software Engineering Institute*. 55 pages. https://resources. sei.cmu.edu/library/asset-view.cfm?assetid=75750, (2003).

[156] Ingram, C., Payne, R., Perry, S., Holt, J., Hansen, F.O., and Couto, L.D.: "Modelling patterns for systems of systems architectures", *IEEE International Systems Conference Proceedings*, pp 146–153. DOI:https://doi.org/10.1109/SysCon.2014.6819249, (2014).

[157] Cuesta, C.E., and Romay, M.P.: "Elements of Self-adaptive Systems – A Decentralized Architectural Perspective", *Springer Berlin. Heidelberg*, Berlin, Heidelberg, pp 1–20. DOI:https://doi.org/10.1007/978-3-642-14412-7_1, (2010).

[158] Romay, M.P., Cuesta, C.E., and Fernández-Sanz, L.: "Self-Adaptation in Systems-of-Systems", *In Proceedings of the 1st International. Workshop on Software Engineering for Systems-of-Systems. IEEE*, Montpellier, France, pp 29–34, (2013).

[159] Ingram, C., Payne, R., and Fitzgerald, J.: "Architectural Modelling Patterns for Systems of Systems", In *INCOSE International. Symposium*, Vol. 25. Wiley Online Library, pp 1177–1192, (2015).

[160] Rothenhaus, K.J., Michael, J.B., and Shing, M. T.: "Architectural Patterns and Auto-Fusion Process for Automated Multisensor Fusion In. SOA System-of-Systems", *IEEE systems journal*, pp 304–316, (2009).

[161] Francalanza, A., Gauci, A., and Pace, G. J., "Distributed System Contract Monitoring", *The Journal of Logic and Algebraic programming*, Vol 82, Issues 5-7, pp 186-215, https://doi.org/10.1016/j.jlap.2013.04.001. (July-October 2013).

[162] Fernandez, E. B., Yoshioka, N., and Washizaki, H., " Two patterns for distributed systems: Enterprise Service Bus (ESB) and Distributed Publish/Subscribe", *Proceedings of the 18th Conference on Pattern Languages of Programs,* Article No.: 8, pp 1–10, https://doi.org/10.1145/2578903.2579146. (October 2011).

[163] Bianchi, T., Santos, D. S., and Felizardo. K. R: "Quality Attributes of Systems-of-Systems: A Systematic Literature Review", *IEEE/ACM 3rd International Workshop on Software Engineering for Systems-of-Systems, Florence*, Italy, DOI: 10.1109/SESoS.2015.12. (17 May 2015).

[164] Guessi, M., Neto, V. V. G., Bianchi, T., Felizardo, K. R., Oquendo, F., and Nakagawa, E. Y.: "A Systematic Literature Review on the Description of Software Architectures for Systems of Systems", *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp 1433-1440, https://doi.org/10.1145/2695664.2695795. (April 2015).

[165] Garces, L., Oquendo, F., and Nakagawa, E. Y.: "Software mediators as first-class entities of systems-of-systems software architectures", *Journal of the Brazilian Computer Society*, Article number 8, (2019).

[166] Santos, D. S., Bianchi, T., Felizardo, K. R., and Nakagawa, E. Y.: "An Investigation on quality attributes of Systems-of-Systems", *In Proceedings Santos AnIO*, (2015).

[167] Gagliardi, M. J., Wood, W. G., Morrow, T., and John, K.: "System of Systems (SoS) Quality Attribute Specification and Architecture Evaluation", *Software Engineering Institute*, (2009).

[168] Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., and Newling, T.: "Patterns: Service-Oriented Architecture and Web Services Book", *IBM International Technical Support Organization*, First Edition, (April 2004).

[169] Garces, L.M.: "A reference architecture for healthcare supportive home systems from a systems-of-systems perspective", *Multiagent Systems*. Thesis, (2018).

[170] Bokhari, S. M. A., and Azam, F.: “Limitations of Service Oriented Architecture and its Combination with Cloud Computing”, *Bahria University Journal of Information & Communication Technologies*, Vol. 8, Issue 1, (April 2015).

Appendix A: Literature Review

Table 57 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 [136]	Santos et al. 2017	Layers	("Big Data" AND ("Software Architecture" OR "Reference Architecture" OR "Reference Model")) [**2017-April 2019**]
S2	Towards a Security Reference Architecture for Big Data [137]	Moreno et al. 2018	----------	
S3	Investigating the Applicability of Architectural Patterns in Big data Systems [73]	Sena et al. 2018	Layers, Broker, Pipes and Filters, Shared-Repository	
S4	A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability [138]	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 [139]	He et al. 2018	----------	
S6	Simplifying Big Data Analytics Systems with a Reference Architecture [140]	Sang et al. 2017	Publish/Subscribe, Broker.	
S7	BlueTalon Data-Centric Security Platform: Bringing Order to Data Security Chaos [141]	Talon, 2016	Layers	Reference of Moreno 2018
S8	Big Data and Data Centric Security [142]	SQRRL, 2014	Layers	Reference of Moreno 2018
S9	A reference web architecture and patterns for real-time visual analytics on large streaming data [143]	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 [144]	Kim et al. 2005	----------	
S11	On the Data Streaming Processing Frameworks: A Case Study [145]	Dhaouadi et al. 2018	----------	
S12	A Reference Architecture for Big Data Systems [146]	Sang, et al. 2016	----------	
S13	A survey on platforms for big data analytics [147]	Singh et al. 2015	----------	
S14	Detecting Irregular Patterns in IoT Streaming Data for Fall Detection [148]	Mahfuz et al. 2018	----------	

S15	Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks [149]	Marcu et al. 2016	----------	("Apache Flink" AND "Pattern")
S16	When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink [150]	Verbitskiy et al. 2016	----------	

Table 58 Primary studies reporting tactics in big data systems

ID	Title	Author(s)/Year	Tactics	Search String
S1	A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability [138]	Carrez et al. 2017	Authentication, Authorization	(("Reference Architecture" OR "Reference Model") AND "Data Streaming System") ("Big Data" AND ("Software Architecture" OR "Reference Architecture" OR "Reference Model"))
S2	Towards a Security Reference Architecture for Big Data [137]	Moreno et al. 2018	Authentication, Authorization, Auditing, Fraud detection, risk control, encryption, Role-based access control. Controlled-access session.	
S3	Simplifying Big Data Analytics Systems with a Reference Architecture [140]	Sang, et al. 2017	Scheduling (Scheduler).	
S4	Research on Reliability Evaluation of Big Data System [151]	Cao et al. 2018	Restart, exception handling, scheduling, authorization	
S5	Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review [152]	Ullah et al. 2018	Performance: ML algorithm optimization, unnecessary data removal, feature selection and extraction, parallel processing, results polling and optimized notification, data cutoff. Accuracy: Alert correlation, combining signature-based and anomaly-based detection, attack detection, algorithm selection, combining multiple detection methods.	

			Scalability: Dynamic load balancing, mapreduce. Reliability: Data ingestion monitoring, maintain multiple copies, dropped netflow detection. Security: security data transmission. Usability: Alert ranking.	
S6	Implementing Reliability: The Interaction of Requirements, Tactics and Architecture Patterns [112]	Harrison et al. 2010	Retry, Restart, Time Stamp, Timeout	

Table 59 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 [72]	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 [12]	Mirakhorli et al. 2016	Security, Reliability, Performance	
S2	A Reference Architecture for Federating IoT Infrastructures Supporting Semantic Interoperability [138]	Carrez et al. 2017	Interoperability	
S3	Towards a Security Reference Architecture for Big Data [137]	Moreno et al. 2018	Security	
S4	Simplifying Big Data Analytics Systems with a Reference Architecture [140]	Sang, et al. 2017	Performance	
S5	Research on Reliability Evaluation of Big Data System [151]	Cao et al. 2018	Reliability	
S6	Architectural Tactics for Big Data Cybersecurity Analytic Systems: A Review [152]	Ullah et al. 2018	Performance, Accuracy, Scalability, Reliability, Security, Usability	

Table 60 Primary studies reporting patterns in the Systems of Ssystems

ID	Title and Reference	Author(s)/Year	Patterns	Search String
S1	Towards an architectural patterns language for Systems-of-Systems [106]	Garces et al. 2019	Service-Oriented Architecture, Enterprise Service Bus, Pipes and Filters, Publish/Subscribe, Reflective Architecture, Pace-Layers, Trickle-Up Software, Contract Monitor, Centralized Architecture, Reconfiguration Control Architecture	("architectural pattern" OR "architectural style") AND ("System-of-Systems" OR "Systems-of-Systems" OR "System of Systems") [2018-October, 2019]
S2	Autonomic Computing Now You See It, Now You Don't. In Software Engineering [153]	Muller et al. 2009	Service-Oriented Architecture	
S3	Evolution styles: Foundations and models for software architecture evolution [154]	Barnes et al. 2014	Service-Oriented Architecture, Enterprise Service Bus	
S4	Understanding Patterns for System-of- Systems Integration Understanding Patterns for System-of- Systems Integration [155]	Kazman et al. 2013	Pipes and Filters, Publish/Subscribe, Enterprise Service Bus, Service-Oriented Architecture,	
S5	Modelling patterns for systems of systems architectures [156]	Ingram et al. 2014	Pipes and Filters, Publish-Subscribe,	
S6	Elements of Self-adaptive Systems – A Decentralized Architectural Perspective [157]	Cuesta et al. 2010	Reflective Architecture	
S7	Self-Adaptation in Systems-of-Systems [158]	Romay et al. 2013	Pace-Layers	
S8	Architectural Modelling Patterns for Systems of Systems [159]	Ingram et al. 2015	Contract Monitor, Reconfiguration Control Architecture	
S9	Architectural Patterns and Auto-Fusion Process for Automated Multisensor Fusion in SOA System-of-Systems [160]	Rothenhaus et al. 2009	Trickle-Up Software pattern	
S10	Distributed System Contract Monitoring [161]	Francalanza et al. 2013	Contract Monitor pattern	
S11	Two patterns for distributed systems: Enterprise Service Bus (ESB) and Distributed Publish/Subscribe [162]	Fernandez et al. 2011	Enterprise Service Bus and Publish/Subscribe	

Table 61 Primary studies reporting tactics in the Systems of Systems

ID	Title	Author(s)/Year	Tactics	Search String
S1	Preserving the Quality of Architectural Tactics in Source Code [11]	Mirakhorlie 2014	Authentication, Authorization	("architectural tactic" AND ("System-of-Systems" OR "Systems-of-Systems" OR "System of Systems")
S2	Implementing Reliability: The Interaction of Requirements, Tactics and Architecture Patterns [113]	Harrison et al. 2010	Retry, Restart, Time Stamp, Timeout	
S3	Towards an architectural patterns language for Systems-of-Systems [106]	Garces et al. 2019	Authentication, Authorization	
S4	Distributed System Contract Monitoring [161]	Francalanza et al. 2013	Load Balancing, Fault tolerance, Pooling	
S5	https://www.sossystems.co.uk/solutions/document-management/	Documentation of systems of systems	Audit Trail	
S6	https://www.sossystems.co.uk/solutions/print-management/	Documentation of systems of systems	Authentication	
S7	https://www.sossystems.co.uk/solutions/print-management/	Documentation of systems of systems	Authorization	

Table 62 Primary studies reporting NFRs in the Systems of Systems

ID	Title	Author(s)/Year	NFR	Search String
S1	Quality Attributes of Systems-of-Systems: A Systematic Literature Review [163]	Bianchi et al. 2015	Performance, Security, Interoperability, Reliability, Safety, Availability, Maintainability, Flexibility, Adaptability, Reusability, Portability, Operability, Integrity, Scalability, Testability, Confidentiality, Usability, Capacity, Fault tolerance, Confort, Sustainability, Effectiveness, Energy, Efficiency, Accountability, Accuracy, Recoverability, Modifiability, Modularity	("Non-Functional Requirement" OR "NFR" OR "Non Functional Requirement" OR "Quality Attribute" OR "QA") AND ("System-of-Systems" OR "Systems-of-Systems" OR "System of Systems")
S2	A Systematic Literature Review on the Description of Software Architectures for Systems of Systems [164]	Guessl et al. 2015	Interoperability, Safety, Integrability, Adaptability, Correctness, Scalability, Security, Maintainability, Performance, Availability, Dependability	

S3	Software mediators as first-class entities of systems-of-systems software architectures [165]	Garces et al 2019	Authorization, Adaptability, Interoperability, Reliability, Scalability, Deployability, Adaptivity, Configurability, Efficiency, Performnace, Compatibility, Maintainability, Security, Portability, Usability.	
S4	An Investigation on quality attributes of Systems-of-Systems [166]	Santose et al. 2015	security, interoperability, performance, reliability and safety	
S5	System of Systems (SoS) Quality Attribute Specification and Architecture Evaluation [167]	Gagliardi et al. 2009	Scalability, Performance, Security, Interoperability, availability, Maintainability, Safety, Usability, Flexibility, Accuracy, Privacy, Reusability, Testability, Extensibility, Reliability	
S6	Detecting, Tracing, and Monitoring Architectural Tactics in Code [12]	Mirakhorli et al. 2016	Security, Reliability, Performance	
S7	Toward an architectural patterns languahe for Systems-of Systems [106]	Garces et al. 2019	Authorization, Adaptability, Interoperability, Reliability,Scalability, Deployability, Adaptivity,Configurability, Efficiency, Performnace, Compatibility, Maaintainability, Security, Portability, Usability.	

Appendix B: GRL Models Inventory

Broker Pattern [84]:

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.

Figure 61 Description of the Broker pattern as documented in [84]

Contribution Values:

Table 63 Contribution values of the Broker pattern

	LocTra	Eff	Port	Main	Scala	Perf	Chang	Exten	Flex	Relia	Reus	Interop
Broker	100	-50	100	0	25	-6	80	100	100	-100	80	100

GRL Model:

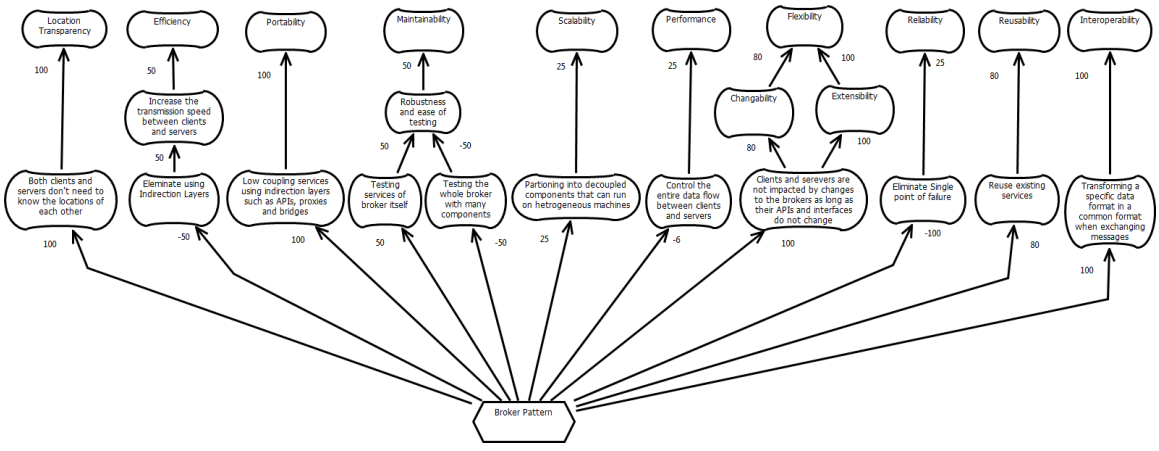


Figure 62 GRL model of the Broker pattern

Evidence Table:

Table 64 Evidence of subgoals' traceability in the Broker pattern model

Broker Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Location Transparency	Both clients and servers don't need to know the locations of each other by using a unique ID	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	Consequences section
Efficiency	The transmission speed between clients and servers by using indirection layers	Applications using a BROKER implementation are usually slower than applications whose COMPONENT distribution is static and known	Consequences section
		BROKER's tasks include locating the appropriate SERVER, forwarding the request to the SERVER and transmitting results and exceptions back to the CLIENT	Solution section
Portability	Low coupling services using indirection layers such as APIs, proxies and bridges	The BROKER SYSTEM hides operating system and NETWORK SYSTEM details from CLIENTS and SERVERS by using indirection layers such as APIs, proxies and bridges	Consequences section
Performance	Control the entire data flow between clients and servers	SYSTEMS that depend directly on a concrete mechanism for inter-process communication also give better performance than a BROKER architecture	Consequences section
Reliability	Single point of failure	BROKER fails during program execution. All the applications that depend on the SERVER or BROKER are unable to continue successfully	Consequences section

Reusability	Reuse existing services	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.	Consequences section
Interoperability	Transforming a specific data format in a common format when exchanging 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	Consequences section

Pipes and Filters Pattern [84]:
This pattern is suitable for processing data streams. It divides the task into several steps and executes these step in a specific order to complete the correct data processing.
Context:
Some applications process streams of data: input data streams are transformed stepwise into output data streams. However, using common and familiar request/response semantics for structuring such types of application is typically impractical. Instead I must specify an appropriate data flow model for them.

Problem:
Modeling a data-flow-driven application raises some non-trivial developmental and operational challenges. First, the parts of the application should correspond to discrete and distinguishable actions on the data flow. Second, some usage scenarios require explicit access to intermediate yet meaningful results. Third, the chosen data flow model should allow applications to read, process, and write data streams incrementally rather than wholesale and sequentially so that throughput is maximized. The long-duration processing activities must not become a performance bottleneck.

Solution:
Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers. Implement each processing step as a separate filter component that consumes and delivers data incrementally, and chain the filters such that they model the application's main data flow. In the data processing pipeline, data that is produced by one filter is consumed by its subsequent filter. Adjacent filters are decoupled using pipes that buffer data exchanged between the filter.

Consequences:
The Pipes and Filters architectural pattern has the following **benefits:**

No intermediate files necessary, but possible. Computing results using separate programs is possible without pipes, by storing intermediate results in files. This approach clutters directories, and is error-prone if you have to set up your processing stages every time you run your system. In addition, it rules out incremental and parallel computation of results. Using Pipes and Filters removes the need for intermediate files, but allows you to investigate intermediate data by using a T-junction in your pipeline.

Flexibility by filter exchange. Filters have a simple interface that allows their easy exchange within a processing pipeline. Even if filter components call each other directly in a push or pull fashion, exchanging a filter component is still straightforward.

Flexibility by recombination. This major benefit, combined with the reusability of filter components, allows you to create new processing pipelines by rearranging filters or by adding new ones. A pipeline without a data source or sink can be embedded as a filter within a larger pipeline.

Reuse of filter components. Support for recombination leads to easy reuse of filter components. Reuse is further enhanced if you implement each filter as an active component, while the underlying platform and shell allow easy end-user construction of pipelines.

Rapid prototyping of pipelines. The preceding benefits make it easy to prototype a data-processing system from existing filters. After you have implemented the principal system function using a pipeline you can optimize it incrementally. You can do this, for example, by developing specific filters for time-critical processing stages, or by re-implementing the pipeline using more efficient pipe connections. Your prototype pipeline can however be the final system if it performs the required task adequately. Highly-flexible filters such as the UNIX tools sed and **awk** reinforce such a prototyping approach.

Efficiency by parallel processing. It is possible to start active filter components in parallel in a multiprocessor system or a network. If each filter in a pipeline consumes and produces data incrementally they can perform their functions in parallel.

Applying the Pipes and Filters pattern imposes some **liabilities:**

Sharing state information is expensive or inflexible. If your processing stages need to share a large amount of global data, applying the Pipes and Filters pattern is either inefficient or does not provide the full benefits of the pattern.

Efficiency gain by parallel processing is often an illusion. This is for several reasons:

- The cost for transferring data between filters may be relatively high compared to the cost of the computation carried out by a single filter. This is especially true for small filter components or pipelines using network connections.

- Some filters consume all their input before producing any output, either because the task, such as sorting, requires it or because the filter is badly coded, for example by not using incremental processing when the application allows it.

- Context-switching between threads or processes is generally an expensive operation on a single-processor machine.

- Synchronization of filters via pipes may stop and start filters often, especially when a pipe has only a small buffer.

Data transformation overhead. Using a single data type for all filter input and output to achieve highest flexibility results in data conversion overheads. Consider a system that performs numeric calculations and uses UNIX pipes. Such a system must convert ASCII characters to real numbers, and vice-versa, within each filter. A simple filter, such as one that adds two numbers, will spend most of its processing time doing format conversion.

Error handling. Error handling is the Achilles' heel of the Pipes and Filters pattern. You should at least define a common strategy for error reporting and use it throughout your system. A concrete error-recovery or error-handling strategy depends on the task you need to solve. If your intended pipeline is used in a 'mission-critical' system and restarting the pipeline or ignoring errors is not possible, you should consider structuring your system using alternative architectures such as Layers.

Figure 63 Description of the Pipes and Filters pattern as documented in [84]

Contribution Values:

Table 65 Contribution values of the Pipes and Filters pattern

	Performance	Flexibility	Reusability	Efficiency	Reliability
Pipes and Filters	16	80	75	50	-100

As I can see in Table 65, the contribution value of the Pipes and Filters pattern with the Performance is **(16)**. This is because in [113], the calculated contribution value is **(0.3125)**. Based on our mathematical calculations, it would be **(16)**. In [111], the contribution value between Pipes and Filters pattern and the Reliability is "- -" and "Key Liability" in [107]. So, it would be **(-100)** in the GRL. The contribution value with the Reusability in [109] is **(1.5)** so it would be **(75)** in the GRL based on our mathematical calculations. While it is **(1.6)** with the Extensibility and Changeability so it would be **(80)** in the GRL. Consequently, it would be **(80)** with the Flexibility. In [111], the contribution value with the Efficiency is "+". Based on our understanding of the description of the Pipes and Filters pattern in [84], I could determine the value to be **(50)**. This is because that the Efficiency is determined as both one of the benefits and liabilities of the Pipes and Filters pattern. This

means the Pipes and Filters pattern has **(+50)** and **(-50)** values with the Efficiency. Because of the "+" in [111], I assigned **(+50)** to the link with the Efficiency.

GRL model:

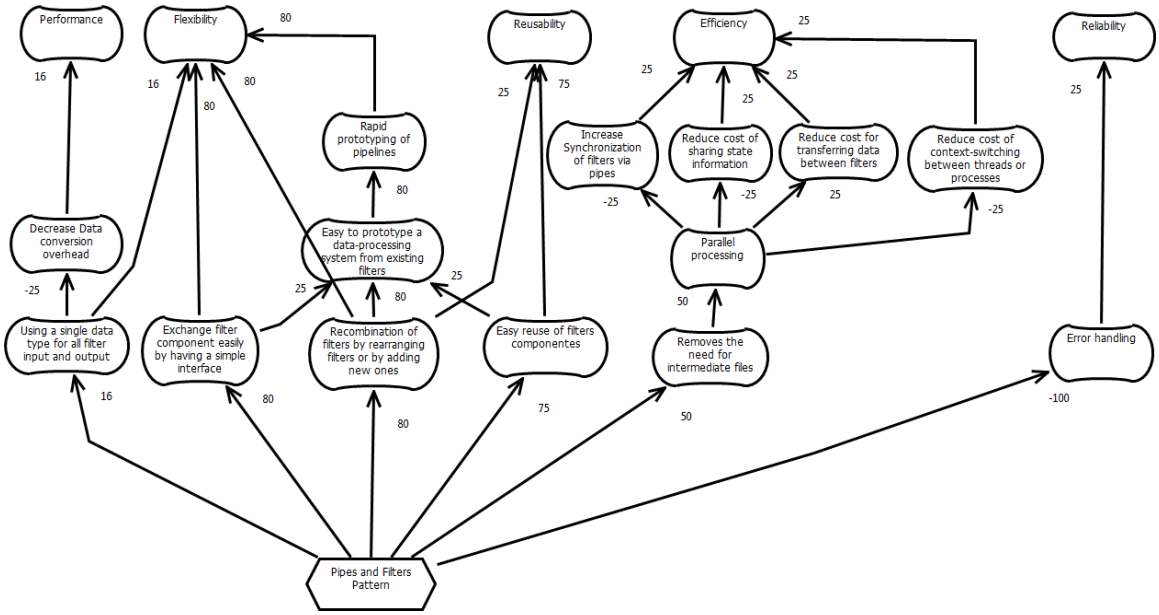


Figure 64 GRL model of the Pipes and Filters pattern

Evidence Table:

Table 66 Evidence of subgoals' traceability in the Pipes and Filters pattern model

Pipes and Filters Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Performance	Using a single data type for all filter input and output	Using a single data type for all filter input and output to achieve highest flexibility results in data conversion overheads.	Consequences Section
	Data conversion overhead		
Flexibility	Using a single data type for all filter input and output	Filters have a simple interface that allows their easy exchange within a processing pipeline.	

	Recombination of filters by rearranging filters or by adding new ones	*Flexibility by recombination:* allows you to create new processing pipelines by rearranging filters or by adding new ones.	
Reusability/ Flexibility	Easy reuse of filters components	*Reuse of FILTER components:* recombination leads to easy reuse of filter components	
Flexibility	Easy to prototype a data-processing system from existing filters	*Rapid prototyping of pipelines:* easy to prototype a data-processing system from existing filters	
Efficiency	Removes the need for intermediate files	Using Pipes and Filters removes the need for intermediate files	
	Synchronization of filters via pipes	Synchronization of filters via pipes may stop and start filters often, especially when a pipe has only a small buffer	
	Cost of sharing state information	*Sharing state information is expensive or inflexible.* If your processing stages need to share a large amount of global data, applying the Pipes and Filters pattern is either inefficient or does not provide the full benefits of the pattern	
Efficiency	The cost for transferring data between filters	The cost for transferring data between filters may be relatively high compared to the cost of the computation carried out by a single filter	
	Cost of context-switching between threads or processes	Context-switching between threads or processes is generally an expensive operation on a single-processor machine.	
Reliability	Error handling	Error handling. Error handling is the Achilles' heel of the Pipes and Filters pattern	

Layers Pattern [84]:
This pattern supports the independent development and evolution of different system parts. It defines one or more layers for the software under development, such that each layer having specific responsibility.

Context:
Regardless of the interactions and coupling between different parts of a software system, there is a need to develop and evolve them independently, for example due to system size and time to- market requirements. However, without a clear and reasoned separation of concerns in the system's software architecture, the interactions between the parts cannot be supported appropriately, nor can their independent development.

Problem:
The challenge is to find a balance between a design that partitions the application into meaningful, tangible parts that can be developed and deployed independently, but does not lose itself in a myriad of detail so that the architecture vision is lost and operational issues such as performance and scalability are not addressed appropriately. An ad hoc, monolithic design is not a feasible way to resolve the challenge. Although it allows quality of service aspects to be addressed more directly, it is likely to result in a spaghetti structure that degrades developmental qualities such as comprehensibility and maintainability.

Solution:
Define one or more layers for the software under development, with each layer having a distinct and specific responsibility. Assign the functionality of the system to the respective layers, and let the functionality of a particular layer only build on the functionality offered by the same or lower layers. Provide all layers with an interface that is separate from their implementation, and within each layer program using these interfaces only when accessing other layers.

Consequences:
The Layers pattern has several **benefits**:

Reuse of layers. If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts. However, despite the higher costs of not reusing such existing layers, developers often prefer to rewrite this functionality. They argue that the existing layer does not fit their purposes exactly, layering would cause high performance penalties-and they would do a better job anyway. An empirical study hints that black-box reuse of existing layers can dramatically reduce development effort and decrease the number of defects.

Support for standardization. Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces. Different implementations of the same interface can then be used interchangeably. This allows you to use products from different vendors in different layers.

Dependencies are kept local. Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed. Changes of the hardware, the operating system, the window system, special data formats and so on often affect only one layer, and you can adapt affected layers without altering the remaining layers. This supports the portability of a system. Testability is supported as well, since you can test particular layers independently of other components in the system.

Exchangeability. Individual layer implementations can be replaced by semantically-equivalent implementations without too great an effort. If the connections between layers are hard-wired in the code, these are updated with the names of the new layer's implementation.

Hardware exchanges or additions are prime examples for illustrating exchangeability. A new hardware 1/O device, for example, can be put in operation by installing the right driver program-which may be a plug-in or replace an old driver program. Higher layers will not be affected by the exchange. A transport medium such as Ethernet could be replaced by Token Ring. In such a case, upper layers do not need to change their interfaces, and can continue to request services from lower layers as before. However, if you want to be able to switch between two layers that do not match closely in their interfaces and services, you must build an insulating layer on top of these two layers. The benefit of exchangeability comes at the price of increased programming effort and possibly decreased run-time performance.

The Layers pattern also imposes **liabilities:**

Cascades of changing behavior. A severe problem can occur when the behavior of a layer changes. Assume for example that I replace a 10 Megabit/sec Ethernet layer at the bottom of our networked application and instead put IP on top of 155 Megabit/sec ATM~. Due to limitations with 1/O and memory performance, our local-end system cannot process incoming packets fast enough to keep up with ATM's high data rates. However, bandwidth-intensive applications such as medical imaging or video conferencing could benefit from the full speed of ATM. Sending multiple data streams in parallel is a high-level solution to avoid the above limitations of lower levels. Similarly, IP routers, which forward packets within the Internet, can be layered to run on top of high-speed ATM networks via multi-CPU systems that perform IP packet processing in parallel.

In summary, higher layers can often be shielded from changes in lower layers. This allows systems to be tuned transparently by collapsing lower layers and/or replacing them with faster solutions such as hardware. The layering becomes a disadvantage if you have to do a substantial amount of rework on many layers to incorporate an apparently local change.

Lower efficiency. A layered architecture is usually less efficient than, say, a monolithic structure or a 'sea of objects'. If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate

layers, and may be transformed several times. The same is true of all results or error messages produced in lower levels that are passed to the highest level. Communication protocols, for example, transform messages from higher levels by adding message headers and trailers.

Unnecessary work. If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance. Demultiplexing in a communication protocol stack is an example of this phenomenon. Several high-level requests cause the same incoming bit sequence to be read many times because every high-level request is interested in a different subset of the bits. Another example is error correction in file transfer. A general purpose low-level transmission system is written first and provides a very high degree of reliability, but it can be more economical or even mandatory to build reliability into higher layers, for example by using checksums.

Difficulty of establishing the correct granularity of layers. A layered architecture with too few layers does not fully exploit this pattern's potential for reusability, changeability and portability. On the other hand, too many layers introduce unnecessary complexity and overheads in the separation of layers and the transformation of arguments and return values. The decision about the granularity of layers and the assignment of tasks to layers is difficult, but is critical for the quality of the architecture. A standardized architecture can only be used if the scope of potential client applications fits the defined layers.

Figure 65 Description of the Layers pattern as documented in [84]

Contribution Values:

Table 67 Contribution values of the Layers pattern

	Reusability	**Testability**	**Portability**	**Changeability**	**Performance**	**Efficiency**
Layers	90	90	95	80	6	-80

As I can see in Table 67, the contribution value of the Layers pattern with the Performance is **(6)**. This is because in [113], the calculated contribution value is **(0.125)**. Based on our mathematical calculations, it would be **(6)**. The contribution value with the Reusability and Testability in [109] is **(1.8)** so it would be **(90)** in the GRL based on our mathematical calculations. The contribution value with the Portability is **(1.9)** in [109], "+" in [111], and "Strength" in [107]. The description of the Layers pattern in [84] shows that the portability can promoted by the existence of the standardized interface so you can adapt affected layers without altering the remaining layers. At the same time, the Portability can be reduced if a layered architecture has too few layers. Since the value in [109] is calculated as **(1.9)**, I assign **(95)** to the contribution value with Portability. The contribution value with the Changeability is **(1.6)** in [109]. The description in [84] shows that the Changeability might

be reduced if a layered architecture has too few layers. However, it is still promoted by the layers pattern based on [109]. Consequently, I assign **(80)** to the link with the Changeability. The contribution value with the Efficiency is "-" in [111]. In the description in [84], it is one of the Layers pattern liabilities. So, I assign **(-80)** to the link with the Efficiency.

GRL model:

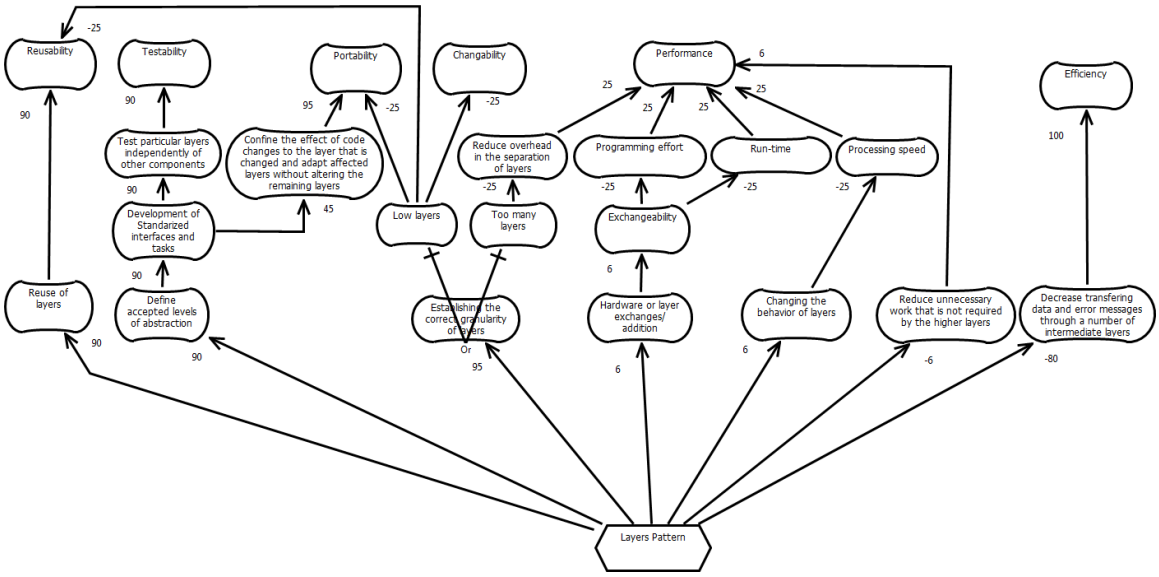


Figure 66 GRL model of the Layers pattern

Evidence Table:

Table 68 Evidence of subgoals' traceability in the Layers pattern model

Layers Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Performance	Processing speed	Our local-end system cannot process incoming packets fast enough to keep up with ATM's high data rates	Consequences section

Reusability	Reuse of layers	If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts	
Testability	Define accepted levels of abstraction	Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces	
	Development of Standardized interfaces and tasks		
	Test particular layers independently of other components	Testability is supported as well, since you can test particular layers independently of other components in the system.	
Testability/Portability	Confine the effect of code changes to the layer that is changed and adapt affected layers without altering the remaining layers	*Dependencies are kept local*. Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.	

Exchangeability	Hardware or layer exchanges/addition		
Performance	Overhead in the separation of layers		
	Programming effort	The benefit of exchangeability comes at the price of increased programming effort and possibly decreased run-time performance	
	Run-time		
	Processing speed	data must be transferred through a number of intermediate layers, and may be transformed several times.	
	Performing unnecessary work that is not required by the higher layers	*Unnecessary work.* If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance.	

	Changing the behavior of layers	*Cascades of changing behavior:* our local-end system cannot process incoming packets fast enough to keep up with ATM's high data rates	
Portability/Changeability/Performance	Establishing the correct granularity of layers	*Difficulty of establishing the correct granularity of layers. A* layered architecture with too few layers does not fully exploit this pattern's potential for reusability, changeability and portability. On the other hand, too many layers introduce unnecessary complexity and overheads in the separation of layers and the transformation of arguments and return values.	

Shared-Repository Pattern [73][85]:
This pattern is a design for applications whose parts operate on, and coordinate their cooperation via, a set of shared data. It consists of using a data repository as communication among different software components crossing domain boundaries.

Context:
Some applications are inherently data-driven: interactions between components do not follow a specific business process, but depend on the data on which they operate. However, despite the lack of a functional means to connect the components of such applications, they must still interact in a controlled manner.

Problem:
A data-driven system operate on massive amounts of data provided by field devices. Core responsibilities like monitoring and control, alarming, and reporting are largely independent of one another, and it is the state of the data that determines the control flow and collaboration of these tasks. Connecting the tasks directly would hard-code a specific business process into the application, which may be inappropriate if specific data is unavailable, not of the required quality, or in a specific state. However, I need a coherent computational state across the entire application.

Solution:
Maintain all data in a central repository shared by all functional components of the data-driven application and let the availability, quality, and state of that data trigger and coordinate the control flow of the application logic. Components work directly on the data maintained by the shared repository, so that other components can react if this data changes. If a component creates new data, or if the application receives new data from its environment, is also stored in the shared repository, to make it accessible to other components.

Forces:
The Shared-Repository pattern allows integration of application functionality with a data-driven control flow to form coherent software systems. It also supports coherent integration of applications that operate on the same data, but neither share nor participate in common business processes. Coordinating components via the state of shared data can introduce performance and scalability bottlenecks, however, if many concurrent components need access to the same data exclusively and are thus serialized. If the shared repository is implemented as a domain object, its concrete implementation is hidden from the application's components and can be swapped or modified transparently. The data maintained by the shared repository is often encapsulated inside managed objects (managed object) that hide the details of concrete data structures and offer meaningful operations for their access and modification. Managed objects allow the application's components to use specific data without becoming dependent on its concrete representation, and support the modification of data representations without effects on the components that use the data.

Figure 67 Description of the Shared-Repository pattern as documented in [73][85]

Contribution Values:

Table 69 Contribution values of the Shared-Repository pattern

	Scalability	Performance	Transparent Modification
Shared-Repository	-50	-50	100

All in the contribution values in Table 69 are extracted from the description of the Shared Repository pattern in [13].

GRL model:

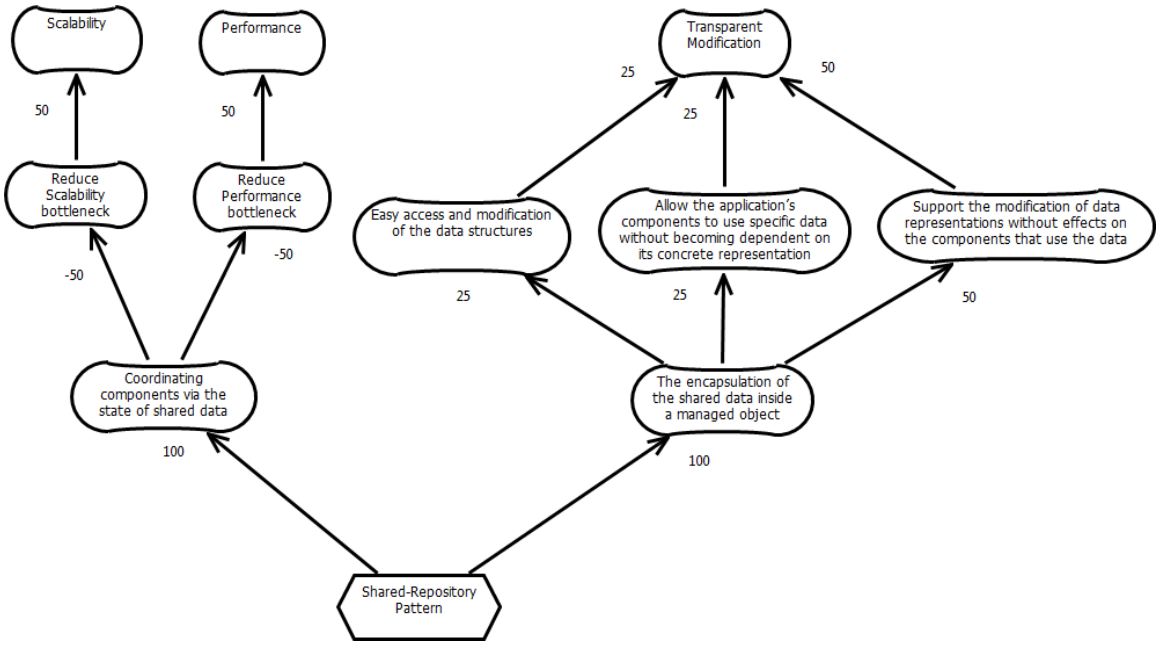


Figure 68 GRL model of the Shared-Repository pattern

Evidence Table:

Table 70 Evidence of subgoals' traceability in the Shared-Repository pattern model

Shared-Repository Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Performance/ Scalability	Coordinating components via the state of shared data		Forces Section
Scalability	Scalability bottleneck		

Performance	Performance bottleneck	Coordinating components via the state of shared data can introduce performance and scalability bottlenecks,	
Transparent Modification	The encapsulation of the shared data inside a managed object	The data maintained by the shared repository is often encapsulated inside managed objects (managed object) that hide the details of concrete data structures and offer meaningful operations for their access and modification	
	Easy access and modification of the data structures		
	Allow the application's components to use specific data without becoming dependent on its concrete representation	allow the application's components to use specific data without becoming dependent on its concrete representation	
	Support the modification of data representations without effects on the components that use the data	support the modification of data representations without effects on the components that use the data.	

Observer/Publish-Subscribe Pattern [75][85]:
This pattern is a messaging pattern where SENDERS of messages categorize published messages into classes without knowledge of which RECEIVERS called subscribers. It allows application components to notify each other about events of interest.

Context:
Components in some distributed applications are loosely coupled and operate largely independently. If such applications need to propagate information to some or all of their components, however, a notification mechanism is needed to inform the components about state changes or other interesting events that affect or coordinate their own computation.

Problem:
Nevertheless, this notification mechanism should not couple application components too tightly, or they will lose their independence. Such application components only want to

know that another component in the system is in a specific state, not which specific component is involved. Similarly, components that disseminate events often do not care which other components want to receive the information. In addition, components should not depend on how other components can be reached, or on their specific location in the system.

Solution:
Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that convey information that may be of interest to others. Notify subscribers interested in those events whenever such information is published. Publishers register with the change propagation infrastructure to inform it about what types of events they can publish. Similarly, subscribers register with the infrastructure to inform it about what types of events they want to receive. The infrastructure uses this registration information to route events from their publishers through the network to interested subscribers. Subscribers receiving events from the infrastructure can use information in the events to guide or coordinate their own computation.

Forces:
Observer/Publish-Subscribe Pattern supports asynchronous communication, in which publishers transmit events to subscribers without blocking to wait for a response. Asynchrony decouples publishers and subscribers so that they can be active and available at different points in time, and also leverages the parallelism inherent in a distributed system. In addition, Observer/Publish-Subscribe allows components in an application to coordinate their computation anonymously without introducing explicit dependencies to one another: they are unaware and independent of each other's location and identity, since they only send and receive events about changes of their state and/or the changed state itself. Observer/Publish-Subscribe also supports group communication, in which publishers of events need not inform each subscriber explicitly, and the infrastructure forwards the events to all interested subscribers. A drawback of anonymous communication is that it can cause unnecessary overhead if subscribers are interested in a specific type of event, and will only react if the event's content meets specific criteria. Publisher/Subscriber can be implemented using fundamental concurrency and network programming patterns [POSA2], resulting in better performance. The information exchanged between the components connected by Observer/Publish-Subscribe middleware is encapsulated inside events, which are realized as messages. An event hides its concrete message format from both the publisher and subscriber(s), as well as from the publisher-subscriber middleware itself, which enables transparent modification of the message's format. Event-driven consumers support the transparent adaptation of a consumer to a specific notification. Such a design enables transparent exchange and evolution of the underlying Observer/Publish-Subscribe infrastructure. A communication failure that cannot be handled internally by the Observer/Publish-Subscribe middleware can be returned as a remote error to the publisher that sent the event.

Figure 69 Description of the Observer/Publish-Subscribe pattern as documented in [75][85]

Contribution Values:

Table 67 Contribution values of the Observer/Publish-Subscribe pattern

	Modifiability	Performance	Availability	Exchangeability
Observer/Publish-Subscribe	100	25	100	75

All in the contribution values in Table 67 are extracted from the description of the Observer/Publish-Subscribe pattern in [13].

GRL model:

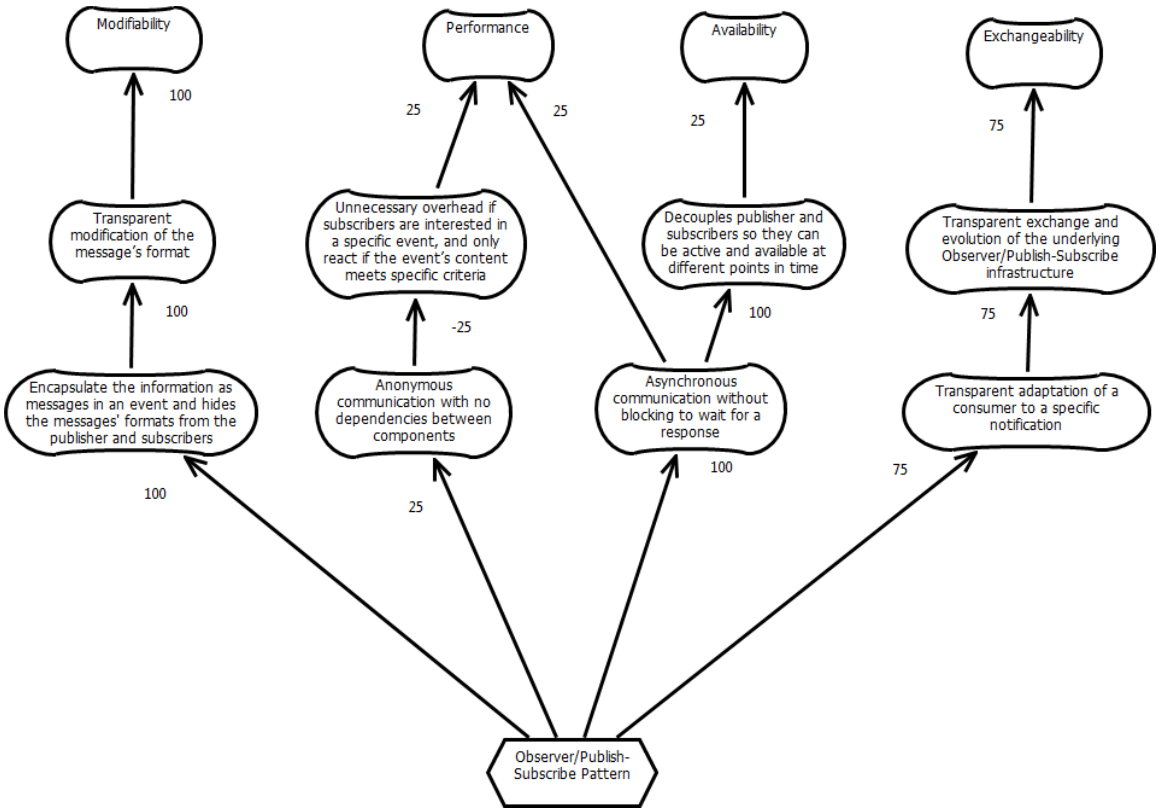


Figure 70 GRL model of the Observer/Publish-Subscribe pattern

Evidence Table:

Table 71 Evidence of subgoals' traceability in the Observer/Publish-Subscribe pattern model

Observer/Publish-Subscribe Pattern			
QA	**Subgoal**	**The underlined Text in the Description**	**The Corresponding Section(s)**
Modifiability	Encapsulate the information as messages in an event and hides the messages' formats from the publisher and subscribers	The information exchanged between the components connected by Observer/Publish-Subscribe middleware is encapsulated inside events, which are realized as messages. An event hides its concrete message format from both the publisher and subscriber(s), as well as from the publisher-subscriber middleware itself, which enables transparent modification of the message's format. Event-driven consumers support the transparent adaptation of a consumer to a specific notification.	Forces Section
	Transparent modification of the message's format		
Performance	Anonymous communication with no dependencies between components	A drawback of anonymous communication is that it can cause unnecessary overhead if subscribers are interested in a specific type of event, and will only react if the event's content meets specific criteria.	
	Unnecessary overhead if subscribers are interested in a specific event, and only react if the event's content meets specific criteria		

Performance/ Availability	Asynchronous communication without blocking to wait for a response	Observer/Publish-Subscribe Pattern supports asynchronous communication, in which publishers transmit events to subscribers without blocking to wait for a response. Asynchrony decouples publishers and subscribers so that they can be active and available at different points in time	
Availability	Decouples publisher and subscribers so they can be active and available at different points in time	Allows components in an application to coordinate their computation anonymously without introducing explicit dependencies to one another: they are unaware and independent of each other's location and identity, since they only send and receive events about changes of their state and/or the changed state itself.	
Exchangeability	Transparent adaptation of a consumer to a specific notification	Such a design enables transparent exchange and evolution of the underlying Observer/Publish-Subscribe infrastructure	
	Transparent exchange and evolution of the underlying Observer/Publish-Subscribe infrastructure		

Availability Tactics (Fault Tolerance Tactics) [70][75][85]:

Ping/Echo refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed ("timed out"). Standard implementations of ping/echo are available for nodes interconnected via IP.

Heartbeat is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check the monitor or the component itself.

Checkpoint/Rollback: Record consistent states and have a path to roll back to them if necessary. Each component in question must define its consistent states and implement a way to record the checkpoints and roll back to them. A component can usually do this without the need for a central component. However, note that the ease of implementation is based on how easy it is to define sane checkpoints – some systems have little notion of state [4].

Time stamp. This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A time stamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Simple sequence numbers can also be used for this purpose, if time information is not important.

Timeout is a tactic that raises an exception when a component detects that it or another component has failed to meet its timing constraints. For example, a component awaiting a response from another component can raise an exception if the wait time exceeds a certain value.

Active redundancy (hot spare). This refers to a configuration where all of the nodes (active or redundant spare) receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, it can take over from a failed component in a matter of milliseconds. The simple case of one active node and one redundant spare node is commonly referred to as 1 + 1 ("one plus one") redundancy. Active redundancy can also be used for facilities protection, where active and standby network links are used to ensure highly available network connectivity.

Exception handling. Once an exception has been detected, the system must handle it in some fashion. The easiest thing it can do is simply to crash, but of course that's a terrible idea from the point of availability, usability, testability, and plain good sense. There are much more productive possibilities. The mechanism employed for exception handling depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception classes that contain information helpful in fault correlation, such as the name of the exception thrown, the origin of the exception, and the cause of the exception thrown. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.

Retry. The retry tactic assumes that the fault that caused a failure is transient and retrying the operation may lead to success. This tactic is used in networks and in server farms where failures are expected and common. There should be a limit on the number of retries that are attempted before a permanent failure is declared.

Escalating restart is a reintroduction tactic that allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.

Figure 71 Description of the Availability tactics as documented in [70][75][85]

Contribution Values:

Table 72 Contribution values of the Availability tactics

	Ping/Echo	Heartbeat	Checkpoint/ Rollback	TimeStamp	Timeout	Active Redundancy	Exception Handling	Retry	Restart
Availability	100	100	100	100	100	100	100	100	100

All in the contribution values in Table 72 are extracted from the description of the availability tactics in [75].

GRL Model:

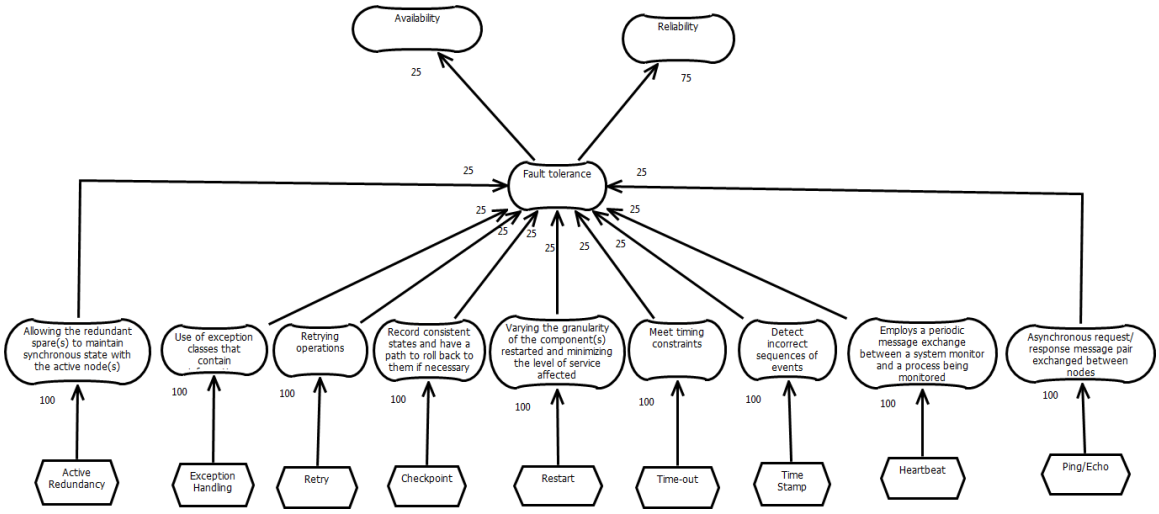


Figure 72 GRL model of the Availability/Reliability tactics

Usability Tactics [75]:

Cancel: When the user issues a cancel command, the system must be listening for it (thus, there is the responsibility to have a constant listener that is not blocked by the actions of whatever is being canceled); the command being canceled must be terminated; any resources being used by the canceled command must be freed; and components that are collaborating with the canceled command must be informed so that they can also take appropriate action.
Usability comprises the following area:
-Minimizing the impact of errors. What can the system do so that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly.

Figure 73 Description of the Usability tactics as documented in [75]

Contribution Values:

Table 73 Contribution values of the Usability tactics

	Cancel
Usability	100

All in the contribution values in Table 73 are extracted from the description of the usability tactics in [75].

GRL Model:

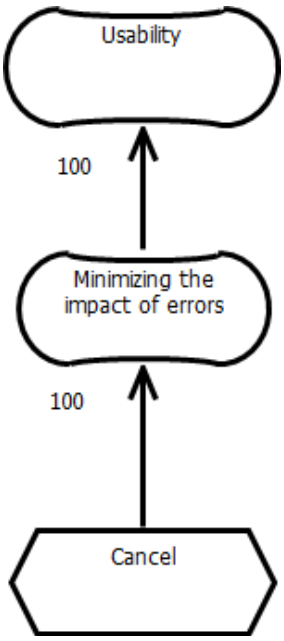


Figure 74 GRL model of the Usability tactics

Security Tactics [75]:

Kerberose: Ensures that a user or a remote system is who it claims to be.

Audit Trail: keep a record of user and system actions and their effects to help trace the actions of, and to identify, an attacker. It supports system recovery and non-repudiation.

Role Based Access Control (RBAC)/ Policy Based Access Control (PBAC): User/Process Authorization is used to ensure that an authenticated user or remote computer/ process has the rights to access and modify either data or services. This tactic is usually implemented through some access control patterns within a system, for example based on user roles/classes or through specific policies.

Session Management: Allows an application to only require the users to authenticate once to confirm that the user requesting a given action is the user who provided the original credentials. This architectural decision will ensure that the authenticated users have a robust and cryptographically secure association with their session.

Authenticate: Authentication means ensuring that an actor (a user or a remote computer) is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, and biometric identification provide a means for authentication.

Figure 75 Description of the Security tactics as documented in [75]

Contribution Values:

Table 74 Contribution values of the Security tactics

	Kerberose	Audit Trail	RBAC/PBAC	Session Management	Authenticate
Security	100	100	100	100	100

All in the contribution values in Table 74 are extracted from the description of the security tactics in [75].

GRL Model:

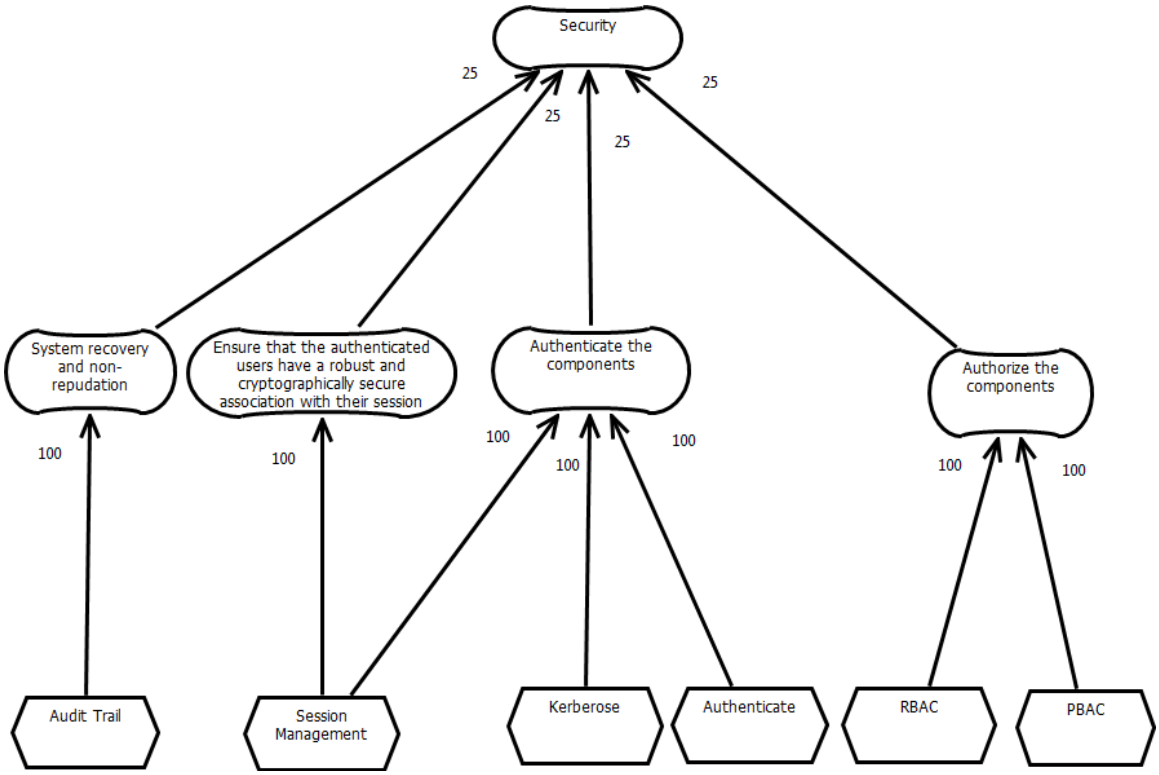


Figure 76 GRL model of the Security tactics

Performance Tactics [75]:

Resource Scheduling: Resource contentions are managed through scheduling policies such as FIFO (First in First out), fixed-priority, and dynamic priority scheduling. Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled.

Resource Pooling: Limited resources are shared between clients that do not need exclusive and continual access to a resource. Pooling is typically used for sharing threads, database connections, sockets, and other such resources.

Load balancing: an application of the scheduling resources tactic. It ensures that one component is not overloaded while another one sits idle.

Figure 77 Description of the Performance tactics as documented in [75]

Contribution Values:

Table 75 Contribution values of the Performance tactics

	Resource Scheduling	Resource Pooling	Load Balancing
Performance	100	100	100

All in the contribution values in Table 75 are extracted from the description of the performance tactics in [75].

GRL Model:

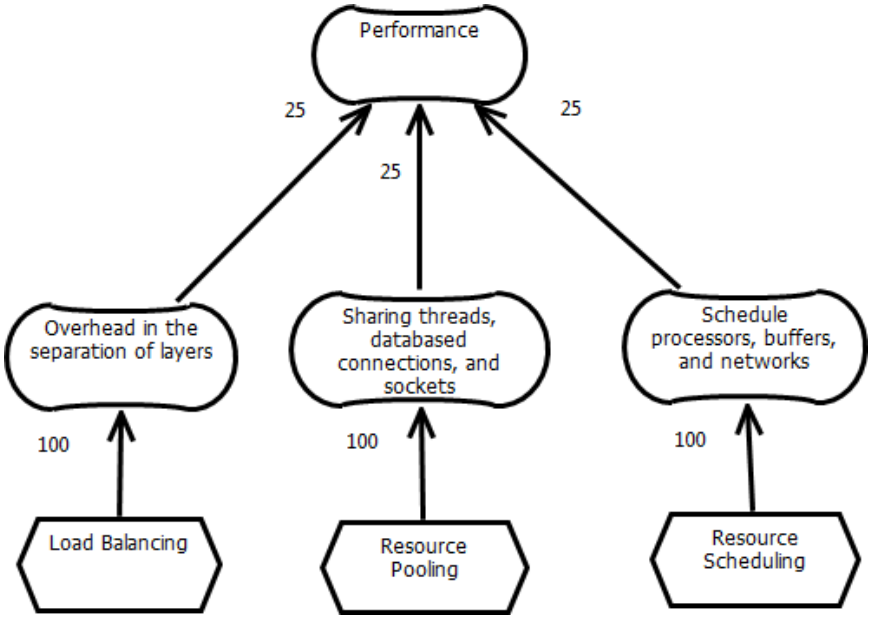


Figure 78 GRL model of the Performance tactics

Case Study 2 (HSH-SoS):

- **Modelling the patterns, tactics, and their contributions on the NFRs**

This step includes the following three sub-steps:

A. Extract NFRs and the Contributions of the Patterns and Tactics to the NFRs from the Descriptions of the Patterns and Tactics

I extract, from the documentation of the patterns and tactics, the NFRs, the contributions of the patterns and tactics on the NFRs, and the design decisions, which show the reason for the negative or the positive impact of a pattern/tactic on an NFR. I consider the descriptions of the patterns as documented in [106][161][162][168][169][170].

Following Ong et al.'s [98] approach, I extract NFRs, design decisions, and the contributions of the patterns of the SoS systems. I added to the description by underlining the benefits, liabilities, NFRs, and reasons for the positive or negative impact of the patterns/tactics on the NFRs. I only focus on the consequences and the solution sections of the description. 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/tactics on the NFRs indicate the design decisions of a pattern/tactic. The design decisions are expressed as phrases starting with an active verb such as define, register, change, reuse, etc.

To illustrate our extraction method, I take the ***Service-oriented Architecture*** (***SOA)*** pattern, as documented in [106][168][169] as an example from the patterns of the SoS systems (see Figure 79). Garces' documentation does not show the liabilities (negative contributions) of ***SOA***. It only shows the benefits and the relative NFRs of the ***SOA*** pattern. The relative NFRs of ***SOA*** are Interoperability, Performance, Reliability, Flexibility, Scalability, Modifiability.

Service-oriented Architecture (SOA). SERVICE-ORIENTED ARCHITECTURE (SOA) gives support to deal with DISTRIBUTEd business PROCESSes. Systems participating in an SOA-based system execute business ACTIVIties and are heterogeneous and under the control of different owners [Josuttis 2007]. SOA pattern promotes interoperability among DISTRIBUTEd systems to accomplish business functionalities (SERVICEs) through the easy integration of their capabilities [Josuttis

2007]. SOA-based system behaviour and performance can be studied by analyzing SERVICE descriptions [Ingram et al. 2015]. SOA also improves reliable solutions, since capabilities can be offered by multiple providers (e.g., constituent systems), hence, if a provider is unavailable another can replace its participation. SOA minimizes the impact of modifications and failures of the SOA-based systems on participant systems, and vice-versa, due this pattern promotes loose-coupling. Moreover, SOA improves flexibility and horizontal scalability, since new systems or systems capabilities can be added to an SOA solution due to standardized interfaces and pre-defined contracts.

Vertical scalability is achieved with the coordination or adaptation of services for creating composed and process SERVICEs, which are high-level SERVICEs defined to execute complex business ACTIVIties work-flows.

Composed and PROCESS SERVICEs are designed using orchestration or choreography approaches. In orchestration, a central CONTROLLER coordinates all PROCESS ACTIVIties. Choreography approach involves collaboration between participants (SERVICEs), which are responsible for executing one or more ACTIVIties. In choreography, no central CONTROLLER exists, hence, collaboration rules must be defined, since SERVICEs are unaware of activities performed by other participants [Josuttis 2007]. Choreography allows better scalability than orchestration, since control can be DISTRIBUTEd among participants, however, it can impact on the system performance, since the full-decentralized control requires to exchange large amounts of information between participants [Garcés et al. 2019b].

Service-oriented architecture benefits [57]:

"As discussed earlier, businesses are dealing with two fundamental concerns: The ability to change quickly, and the need to reduce costs. To remain competitive businesses must adapt quickly to internal factors such as acquisitions and restructuring, or external factors like competitive forces and customer requirements. Cost-effective, flexible IT infrastructure is needed to support the business. With service-oriented architecture, I can realize several benefits to help organizations succeed in the dynamic business landscape of today:"

- Leverage existing assets. SOAs provide a layer of abstraction that enables an organization to continue leveraging its investment in IT by wrapping these existing assets as services that provide business functions. Organizations potentially can continue getting value out of existing resources instead of having to rebuild from scratch.
- Easier to integrate and manage complexity. The integration point in a service-oriented architecture is the service specification and not the implementation. This provides implementation transparency and minimizes the impact when infrastructure and implementation changes occur. By providing a service specification in front of existing resources and assets built on disparate systems, integration becomes more manageable since complexities are isolated. This becomes even more important as more businesses work together to provide the value chain.
- More responsive and faster time-to-market. The ability to compose new services out of existing ones provides a distinct advantage to an organization that has to be agile to respond to demanding business needs. Leveraging existing components and services reduces the time needed to go through the software development life cycle of gathering requirements, performing design, development and testing. This leads to rapid development of new business services and allows an organization to respond quickly to changes and reduce the time-to-market.
- Reduce cost and increase reuse.

  With core business services exposed in a loosely coupled manner, they can be more easily used and combined based on business needs. This means less duplication of resources, more potential for reuse, and lower costs.
- Be ready for what lies ahead. SOAs allows businesses be ready for the future. Business processes which comprise of a series of business services can be more easily created, changed and managed to meet the needs of the time. SOA provides the flexibility and responsiveness that is critical to businesses to survive and thrive.

Service-oriented architecture liabilities [58]:

- Exponential increase in connections: Increases burden on sever and management overhead due to transmission control protocol.

- XML format messaging: Messages are large in size due to which more bandwidth and resources are consumed.
- Reliability: Due to multiple points of failure, it is hard to determine the reliability of each point. Software debugging can affect both client and server side.
- Scalability: It's hard to add a new application or require additional development.

Figure 79 Description of the SOA pattern as documented in [106][168][169]

A. Calculate the contribution values of the patterns and tactics to the NFRs

I used the match method between the relationships between a pattern/tactic and NFRs from the literature and equivalent contribution values in the GRL. I used the scale shown in Table 21 in Chapter 5. That is, if the relationship between a pattern or a tactic and a given NFR is "Key Strength", "Very Strong", or "++", then the contribution value would be **(+100)** in the GRL. While, if the relationship between a pattern or a tactic and a given NFR is "Key Liability", "Very Weak", or "- -", then the contribution value would be **(-100)**. In case that the relationship between a pattern or a tactic and a given NFR is "Neutral", "~" or "=", then the contribution value would be **(0)**. While, if the relationship between a pattern or a tactic and a given NFR is only "Strength", "Strong" or "+", then the contribution value would be any value between 1 and 99 in the GRL. The contribution value will be any value between -1 and -99 if the relationship between a pattern or a tactic and a given NFR is "Liability", "Weak", or "-".

I could determine the contribution values of the patterns or tactics of the HSH-SoS, which have contribution values between 1 and 99 or-1 and -99, based on our understanding of their strength, as shown in their descriptions as documented in [106]. I calculated the contribution values of the Service-oriented Architecture pattern, as shown in Table 76.

Contribution Values:

Table 76 Contribution values of the SOA pattern

	Interoperability	Scalability	Reusability	Performance	Flexibility	Modifiability	Reliability
SOA	100	75	85	-25	75	85	0

As I can see in Table 76, based on our standing of the benefits of the Service-oriented Architecture ***SOA*** pattern, I could determine that it has a positive contribution of type **(85)** with the *Modifiability*. This is because ***SOA*** minimizes the impact of modifications and failures of the SOA-based systems on participant systems, and vice-versa, due this pattern promotes loose-coupling. While it has a positive contribution value of type **(100)** with the interoperability. This is because that the ***SOA*** pattern promotes interoperability among distributed systems to accomplish business functionalities (services) through the easy integration of their capabilities, as discussed in its documentation in [106]. The ***SOA*** pattern also has a negative contribution of type **(-25)** with the Performance. This is because of one of the liabilities of the ***SOA*** pattern is the exponential increase in connections, which increases the burden on sever and management overhead due to transmission control protocol in the ***SOA*** pattern, which decreases the Performance requirement as discussed in the documentation of the ***SOA*** pattern in [106].

B. Derive the Models of the Patterns and Tactics and their Contributions to the NFRs from the Descriptions of the Patterns and Tactics

I derived GRL models of the detected patterns and tactics and their contributions to the NFRs from the description of each pattern/tactic following the general model of a pattern/tactic, which is shown in Figure 11 in Chapter 5.

Regarding the tactics of SoS, as I mentioned before, I could determine that the most common tactics in SoS systems are the same as the tactics in big data systems. So, I already used the models of tactics which are derived in the big data systems example.

Figure 80 shows the derived GRL model of the Service-oriented Architecture ***SOA*** pattern and its contribution to the NFRs. As I can see in Figure 80, the ***SOA*** pattern helps the loose coupling, which pushes the ***SOA*** pattern toward the *Modifiability*. This is because ***SOA*** minimizes the impact of modifications and failures of the SOA-based systems on participant systems, and vice-versa. Therefore, a positive contribution of type **(85)** is shown between ***SOA*** and *Modifiability*. The ***SOA*** pattern also promotes both *Flexibility* and *Scalability*. This is because, in the Service-oriented Architecture, new systems or systems capabilities can be added to an ***SOA*** solution due to standardized interfaces and pre-defined

contracts. Therefore, a positive contribution of type **(75)** is shown between ***SOA*** and both *Flexibility* and *Scalability.* The ***SOA*** pattern also promotes *Interoperability* among distributed systems to accomplish business functionalities through the easy integration of their capabilities. Therefore, a positive contribution of type **(100)** is shown between ***SOA*** and *Interoperability*.

GRL Model:

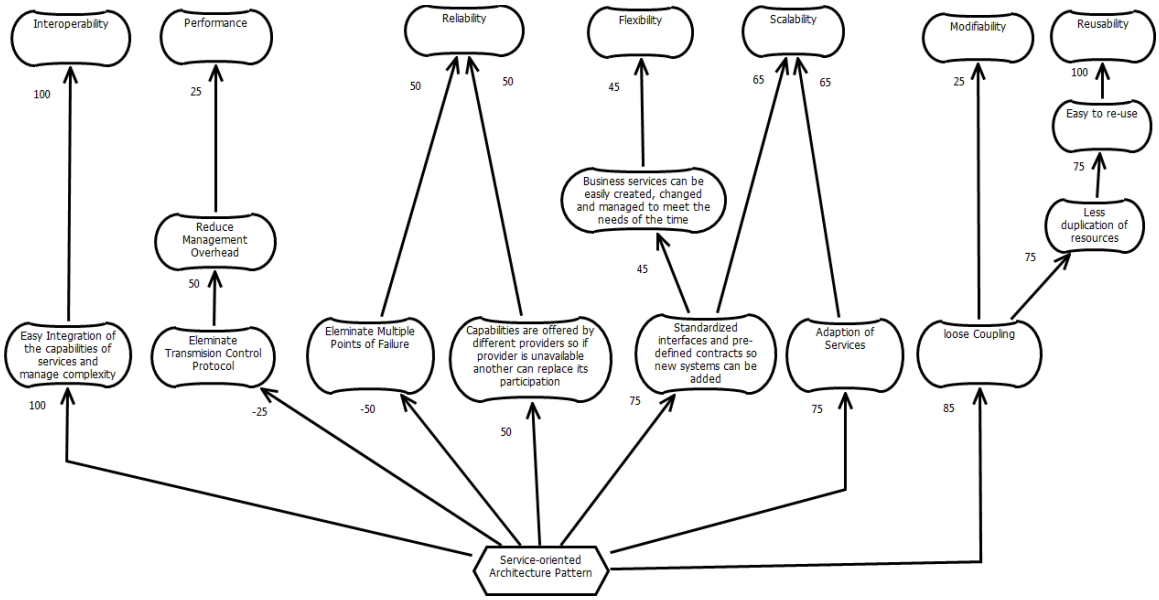


Figure 80 GRL model of the SOA pattern

Enterprise Service Bus [162]

Intent

Provide a convenient infrastructure to integrate a variety of distributed services and related components in a simple way.

Example

A travel agency interacts with many services to do flight reservations, check hotel availability, check customer credit, and others. This interaction is being done now by direct interaction, which results in many ad hoc interfaces, and requires many format conversions. The system is not scalable and it is hard to support standards.

Context

Distributed applications using web services, as well as related services such as directories, databases, security, and monitoring. There may be also other types of components (J2EE, .NET). There may be different standards applying to specific components and components that do not follow any standards.

Problem

When an organization has many scattered services, how can I aggregate them so they can be used together to assemble applications, at the same time keeping the architectural structure as simple as possible, and apply uniform standards?

Solution
Introduce a common bus structure that provides basic brokerage functions as well as a set of other appropriate SERVICEs. One can think of this BUS as an intermediate LAYER of processing that can include SERVICEs to handle problems associated with reliability, scalability, security, and communications disparity. An ESB is typically part of a SERVICE-Oriented Architecture Implementation Framework, which includes the infrastructure needed to implement a SOA system. This infrastructure may also include support for stateful SERVICEs.

Consequences
This pattern provides the following benefits:
• *Interoperability.* The ESB through its architecture and use of adapters provides a way to interact with a variety of services, internal or external.
• *Simplicity of structure*: much simpler than point-to-point or any other interconnection structure.
• *Scalability:* the number of interconnected services can be increased easily.
• *Message flexibility*: I can provide a variety of message invocation styles (synchronous and asynchronous) by using different message patterns.
• *Flexibility:* New types of services can be accommodated easily since they only need to conform to the interface standards.
• *Simplicity of management*: I can centralize the functions of monitoring and management of services, as well as any other needed functions.
• *Transparency*: I can find services conveniently by having lookup services.
• *Quality of service*: by using appropriate associated services I can provide different degrees of security, reliability, availability, or performance.
• *Use of policies*: I can use institution policies for configuration and management.
This allows convenient governance and systematic changes. Security policies can define rights for the users with respect to the services.
• *Standard interfaces*: I can define explicit and formal interface contracts that must be followed by all aggregated functions.

Liabilities include:
• Extra overhead compared to point-to-point, because of the indirection involved and the overhead of the ESB itself.
• The bus is a single point of failure, but this can be overcome using redundancy
• A common interface standard may not be the most convenient for some services.
Some applications may need more functions or parameters to interact with others than the ones defined in the common interfaces. Designing such a common interface may not be easy either.
• The bus may hide component dependencies.

Figure 81 Description of the Enterprise-Service Bus pattern as documented in [162]

Contribution Values:

Table 77 Contribution values of the Enterprise-Service Bus pattern

	Performance	Scalability	Interoperability	Security	Reliability	Availability	Flexibility
Enterprise-Service Bus	-100	25	50	25	-75	-75	50

All in the contribution values in Table 77 are extracted from the description of the Enterprise-Service Bus pattern in [162].

GRL Model:

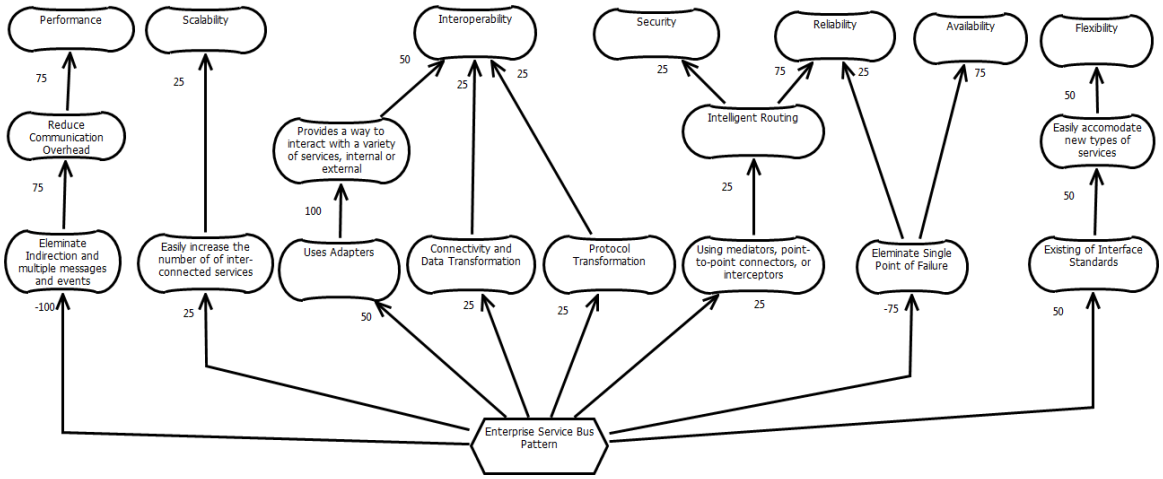


Figure 82 GRL model of the Enterprise Service Bus pattern

Trickle-Up Software Pattern [106]. The TRICKLE-UP pattern is a MULTI-LAYERed pattern that allows separation of concerns for DATA MANAGEMENT. Data object management and fusion logic are independent services that can be binding in runtime to promote control and monitoring of DATA MANAGEMENT. This pattern is principally used when environment data are collected and must be further analyzed and AGGREGATEd to create consolidated reports of a situation. In SoS context, the TRICKLE-UP pattern can be used for knowledge AGGREGATion and discovering required to support SoS emergent behaviors.

Figure 83 Description of the Trickle-Up pattern as documented in [106]

Contribution Values:

Table 78 Contribution values of the Trickle-Up pattern

	Modifiability	Reusability	Security
Trickle-up	25	25	-25

All in the contribution values in Table 78 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

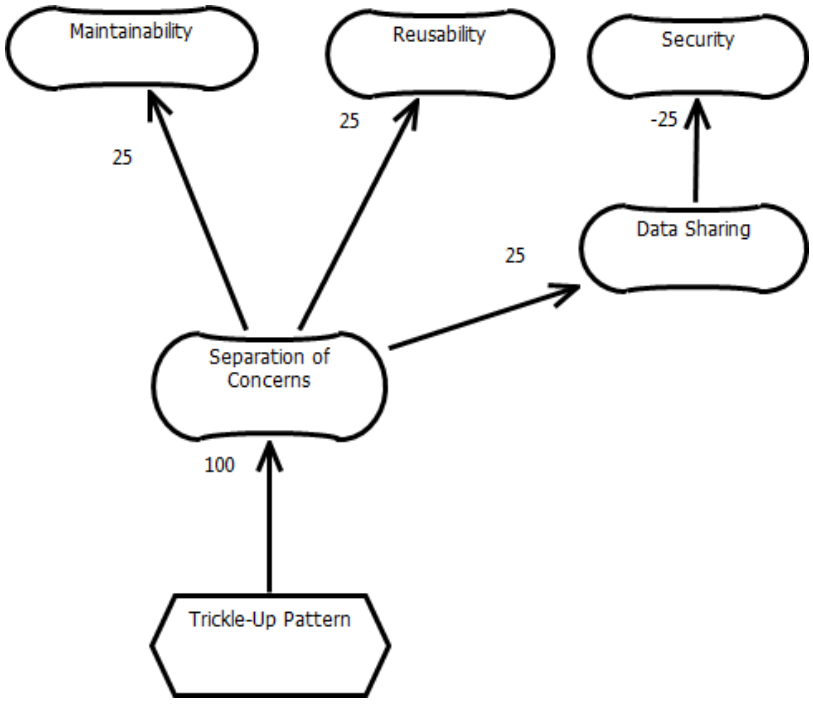


Figure 84 GRL model of the Trickle-Up pattern

Reconfiguration Control Architecture [106]. The aim of this pattern is to promote DYNAMIC RECONFIGURATIONs of software architectures. For this, a RECONFIGURATION CONTROL entity (e.g., a central CONTROLler) is responsible for monitoring constituent systems performance and functionality in the form of metadata obtained from each system. Based on such metadata, the CONTROLler, making use of RECONFIGURATION policies, determines when a RECONFIGURATION is necessary and what actions must be executed [97]. In SoS, this pattern can be used to avoid degradation of SoS missions due to modifications or unavailability of its constituent systems.

Figure 85 Description of the Reconfiguration Control Architecture pattern as documented in [106]

Contribution Values:

Table 79 Contribution values of the Reconfiguration Control Architecture pattern

	Performance	Availability	Modifiability
Reconfiguration Control Architecture	100	-100	25

All in the contribution values in Table 79 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

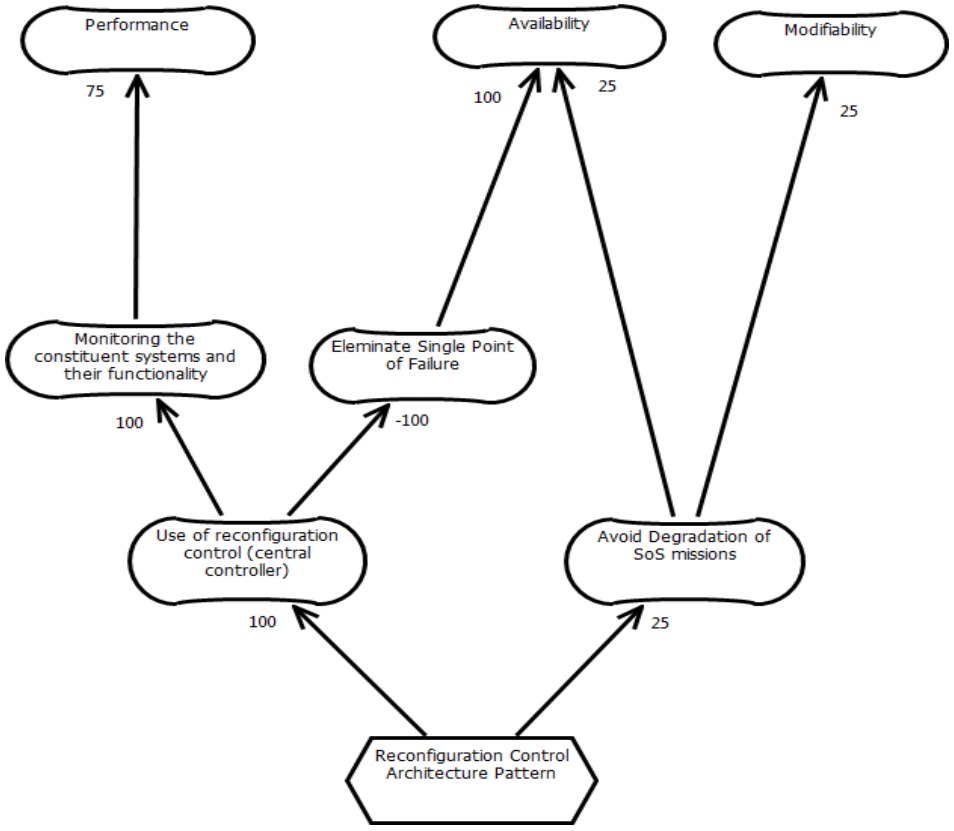


Figure 86 GRL model of the Reconfiguration Control Architecture pattern

Contract Monitor [106]. This pattern aims to monitor constituent systems INTERFACEs in order to identify possible deviations from its expected functionalities that can prejudice expected SoS emergent behaviors. For using this pattern, it is assumed

that constituent systems INTERFACEs are associated to CONTRACTs of behavior, and that composition of such CONTRACTs can be correlated to SoS emergent behaviors. CONTRACT MONITORs can be implemented internally in a constituent system, as an entity, under SoS control, MONITORing constituent systems INTERFACEs, or as an external entity MONITORing interactions between constituent systems to study emergent behaviors [159].

"More and more systems are deployed in a distributed fashion, whether out of our choice or necessity. Distribution poses major design challenges for runtime monitoring of contracts, since monitors themselves can be distributed, and trace analysis can be carried out remotely across location. This impacts directly various aspects of the system being monitored, from the security of sensitive information, to resource management and load balancing, to aspects relating to fault tolerance." [106].

Figure 87 Description of the Contract Monitor pattern as documented in [106][161]

Contribution Values:

Table 80 Contribution values of the Contract Monitor pattern

	Security	Reliability	Performance
Contract Monitor	-100	-100	-100

All in the contribution values in Table 80 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

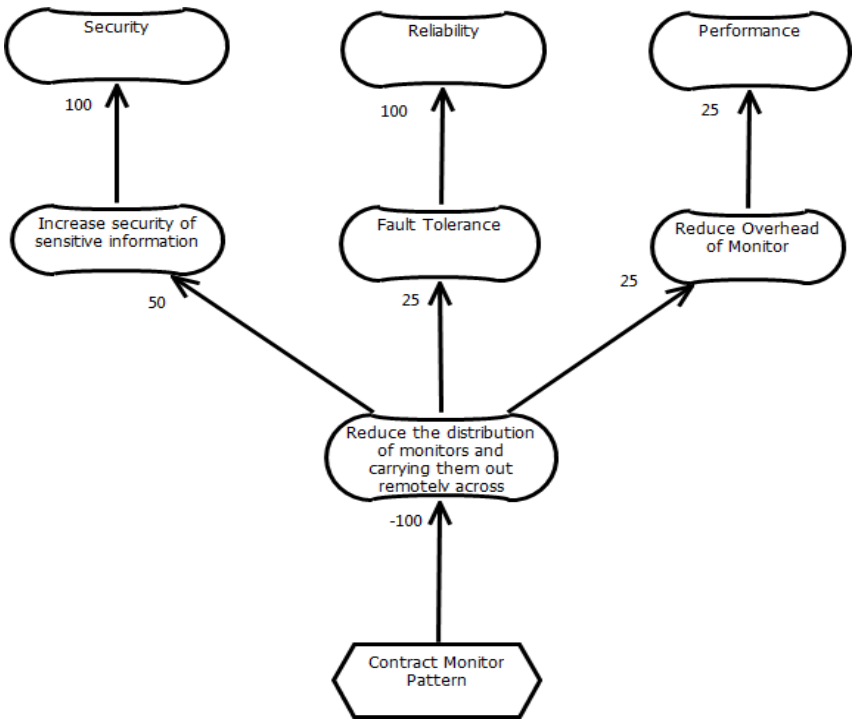


Figure 88 GRL model of the Contract Monitor pattern

Pace-Layers [106]. Also named LAYERs of change, is proposed as an initial strategy to design SoS [158]. LAYERs allow the construction of complex behaviors through LAYERs hierarchies, where lower LAYERs implement fast adaptations (i.e., reconfigurations on constituent systems) and higher LAYERs are responsible for time demanding adaptations (i.e., selection of the best policy or plan to achieve SoS missions based on current system status) [73][158]. Lower LAYERs adaptations aim to achieve performance requirements, and adaptations in higher LAYERs seem to address reliability requirements [73]. Moreover, as LAYERs allow separation of concerns, this property supports maintainability and reusability requirements. Interoperability can also be supported by SoS lower LAYERs at establishing well-defined interfaces of constituent systems.

Figure 89 Description of the Pace-Layers pattern as documented in [106]

Contribution Values:

Table 81 Contribution values of the Pace-Layers pattern

	Performance	Reliability	Maintainability	Reusability	Security	Interoperability
Pace-Layers	50	50	25	25	25	25

All in the contribution values in Table 81 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

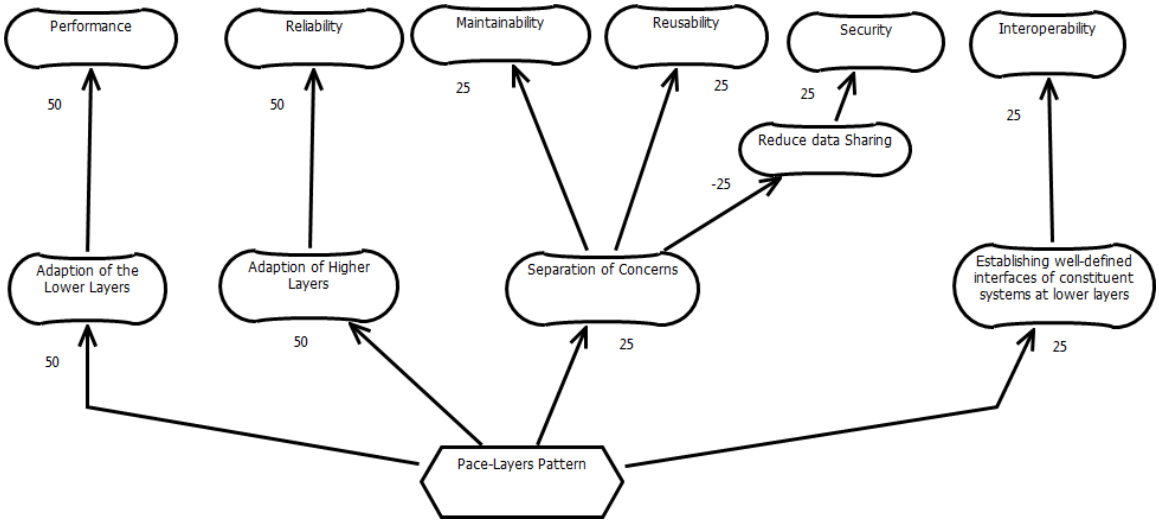


Figure 90 GRL model of the Pace-Layers pattern

Reflective Architecture [106]. REFLECTtion is defined as "the capability of a system to rationalize and act upon itself" [157]. A system with REFLECTtion capability is composed of two LAYERs. A bottom LAYER describing system operations and configuration (e.g., components, interfaces, data, interconnections, etc.), and an upper LAYER embracing an internal meta-model representing how the system perceives and/or modifies itself. Such model contain all structural a behavioral aspects of the system and is separated from the application logic components. Hence, the system will REFELCT changes performed in the model, and vice versa through the execution of two basic operations named REIFICATion and REFLECTion.
REIFICATion is the action to transfer current system status to make alterations in the meta-model. REFLECTion is the operation of change system configuration or behavior regarding modifications in the meta-model. In this perspective, REFLECTion architecture makes the system more flexible, since it can ADAPT itself to changing conditions. Moreover, REFLECTion allows for coping with unforeseen situations automatically. In SoS, the REFLECTion architecture can support evolution and changes in runtime, promoting the anticipation of SoS predicted emergent behaviors and the detection of unforeseen ones.

Figure 91 Description of the Reflective-Architecture pattern as documented in [106]

Contribution Values:

Table 82 Contribution values of the Reflective-Architecture pattern

	Efficiency	**Flexibility**
Reflective-Architecture	-25	50

All in the contribution values in Table 82 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

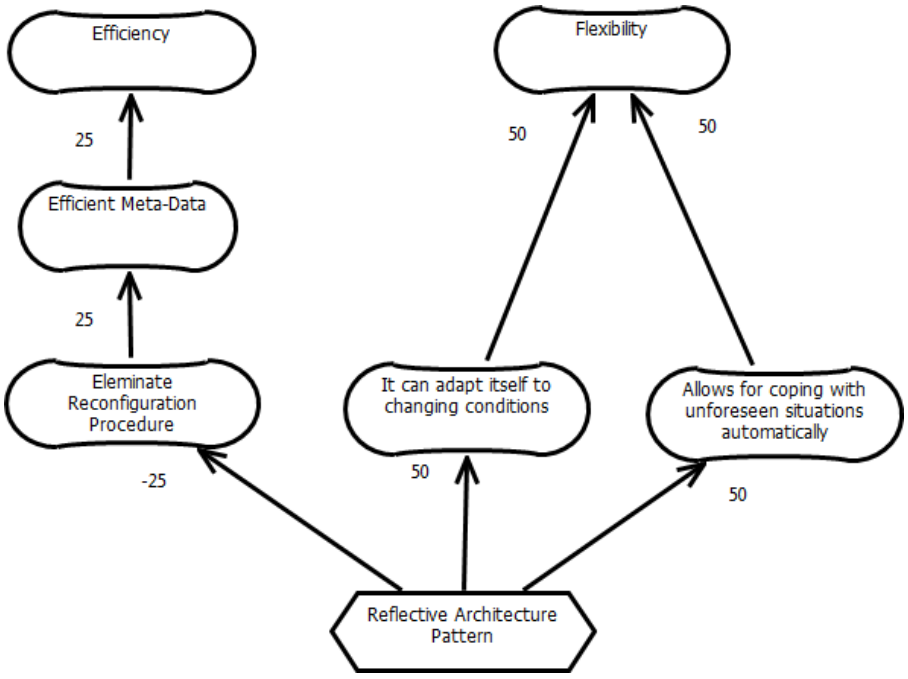


Figure 92 GRL model of the Reflective Architecture pattern

Centralized Architecture [106]. In a CENTRALized architecture, a CENTRAL CONTROLLER, defined as a HUB, is responsible for guarantee the correct behavior and allow the accomplishment of missions of the SoS. CONTROL can be characterized as: (i) fully CENTRALized, when just one CENTRAL CONTROLLER is responsible for meeting SoS goals; (ii) hierarchical CENTRALized, when constituent systems act as CONTROLLERs themselves; (iii) hybrid CENTRALized-DISTRIBUTE, if control\ activities are DISTRIBUTEd among different constituent systems and HUBs.

Figure 93 Description of the Centralized Architecture pattern as documented in [106]

Contribution Values:

Table 83 Contribution values of the Centralized Architecture pattern

	Reliability	Availability
Centralized Architecture	-25	-25

All in the contribution values in Table 83 are extracted from the description of the Observer/Publish-Subscribe pattern in [106].

GRL Model:

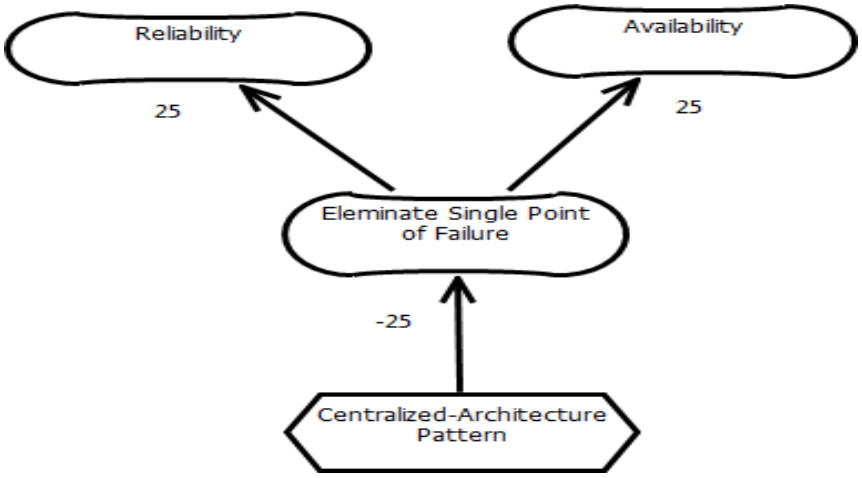


Figure 94 GRL model of the Centralized-Architecture pattern

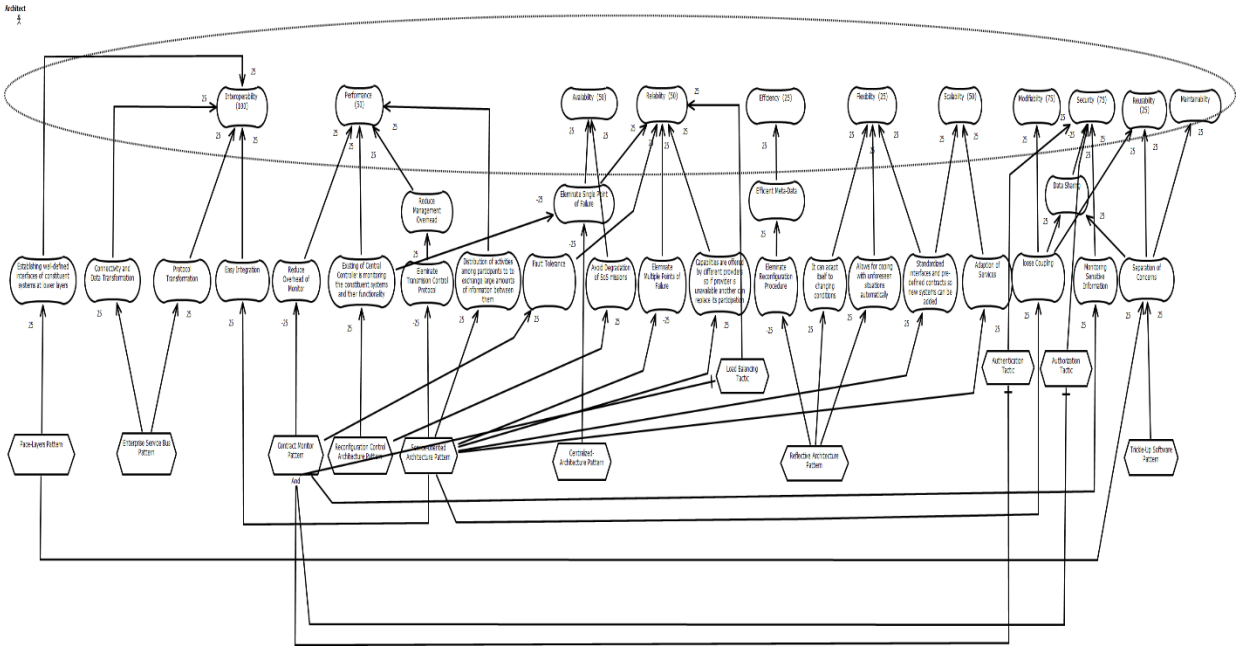


Figure 95 GRL model of the HSH-SoS architecture

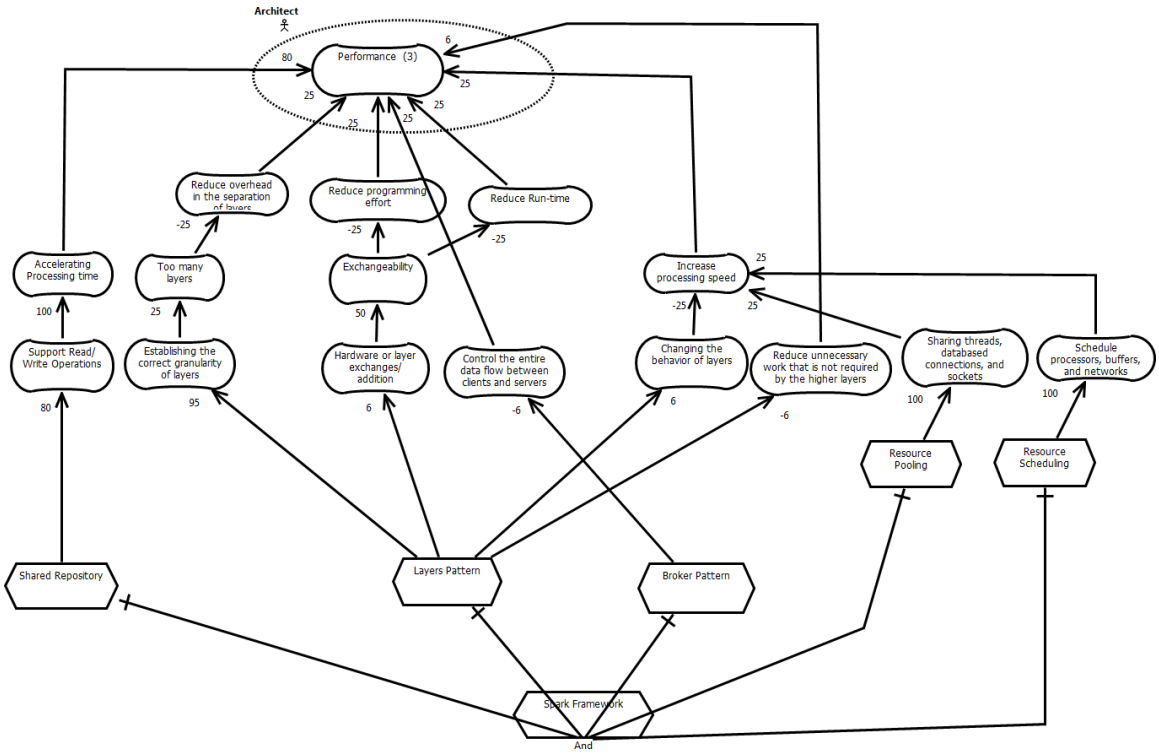


Figure 96 GRL model of Spark in terms of Performance

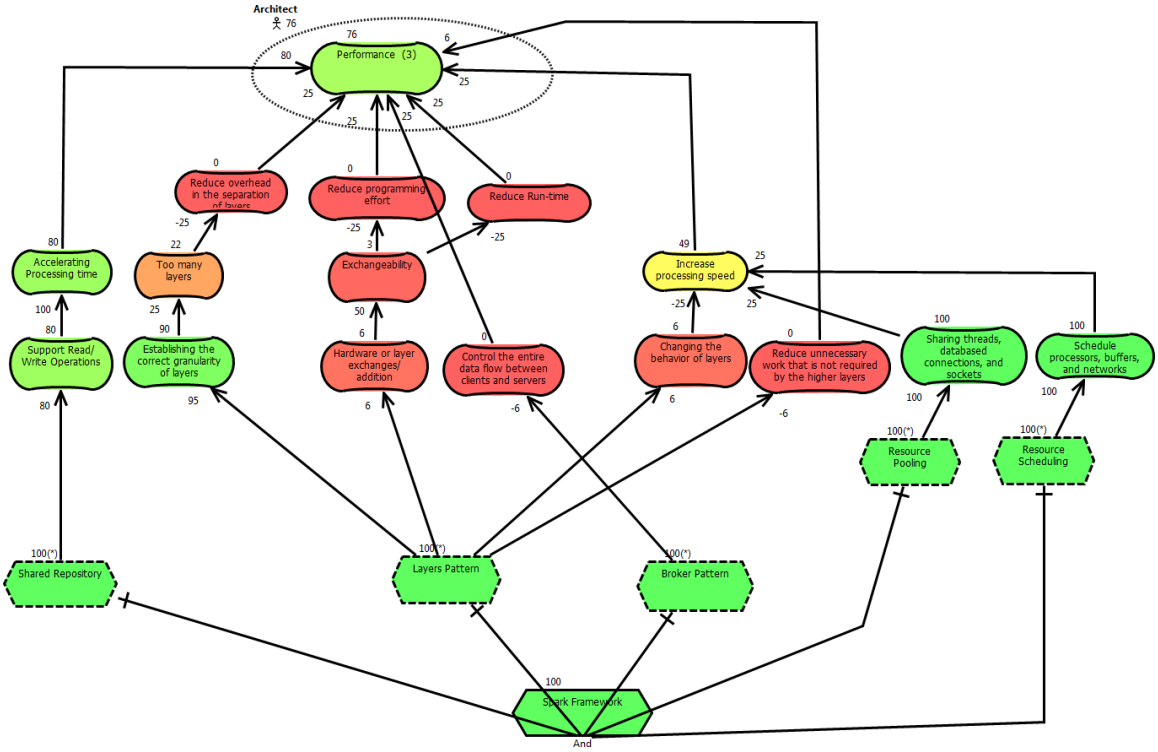


Figure 97 Evaluated GRL model of Spark in terms of Performance

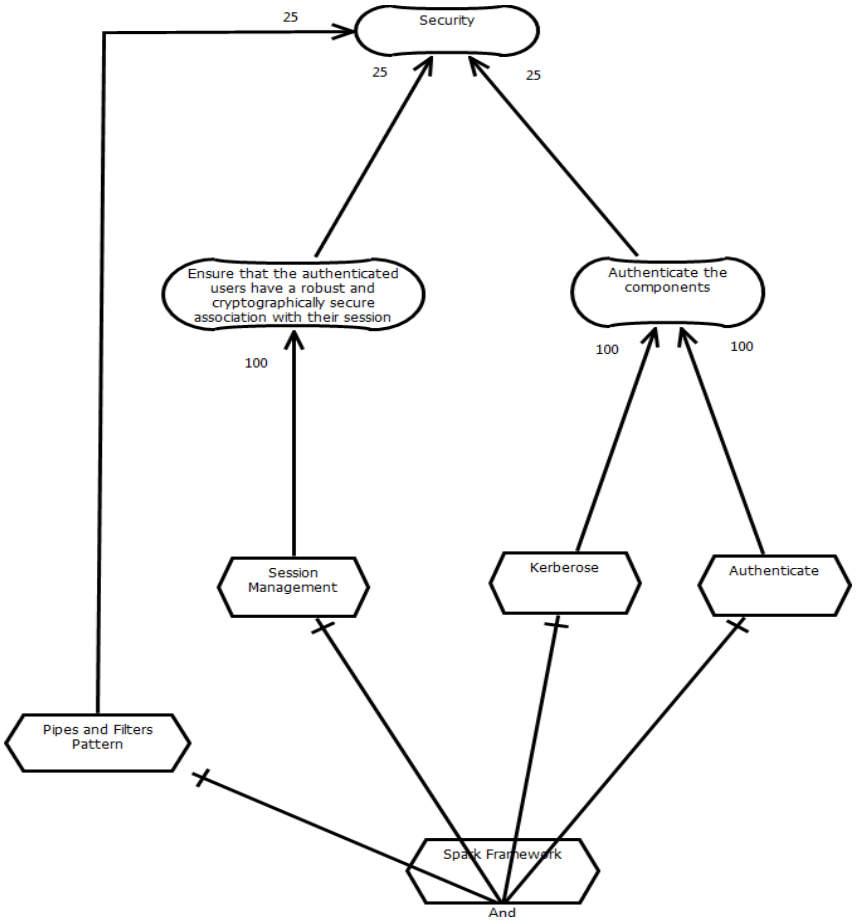


Figure 98 GRL model of Spark in terms of Security

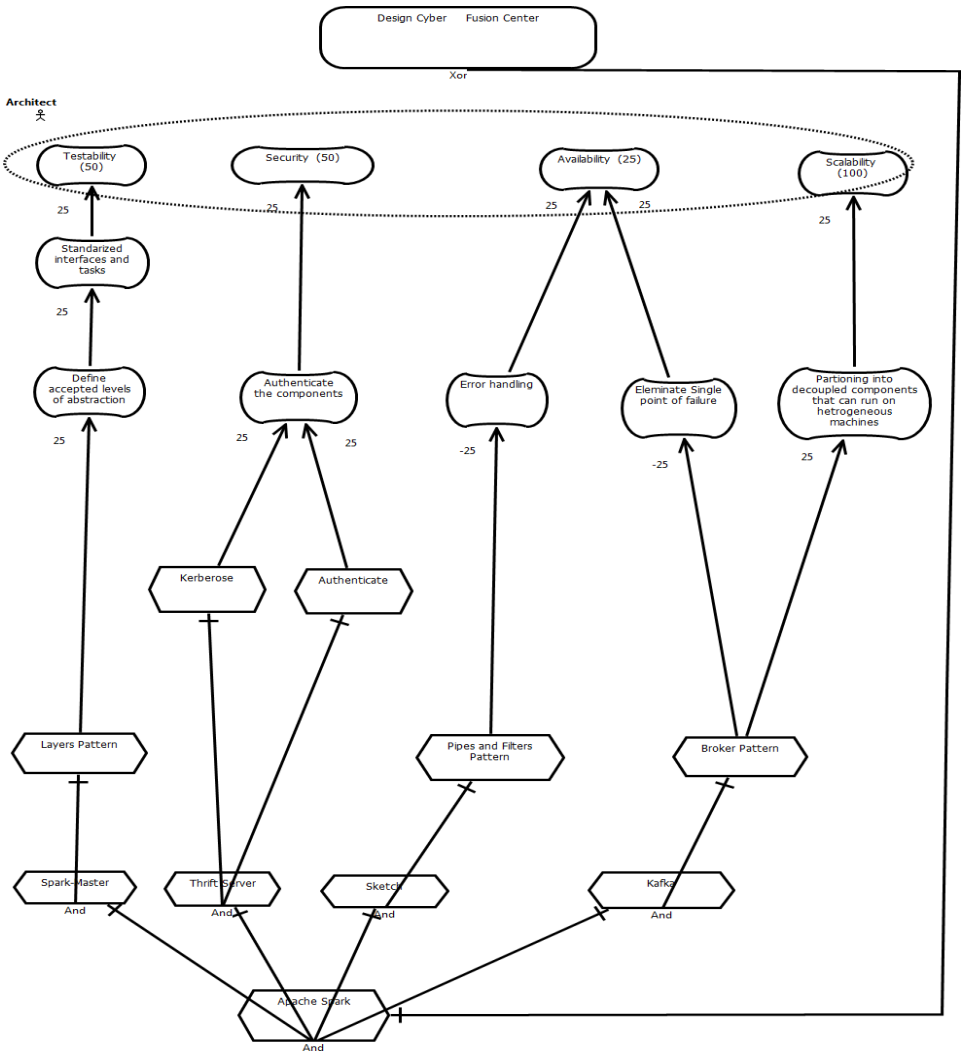


Figure 99 GRL model of Spark in terms of Testability, Security, Availability, and Scalability

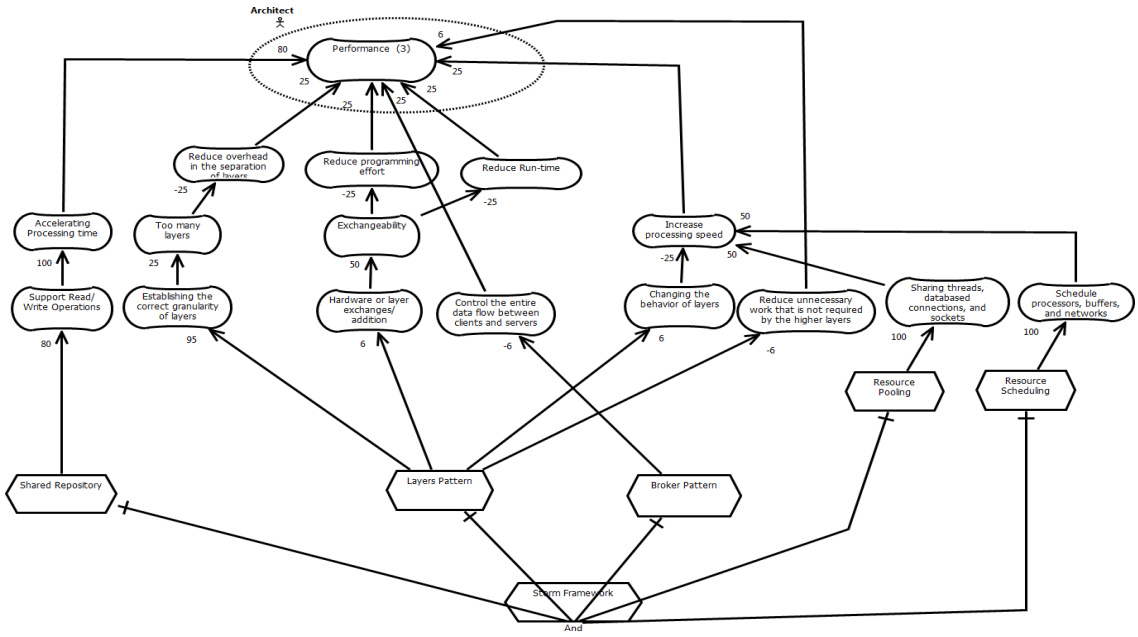


Figure 100 GRL model of Storm in terms of Performance

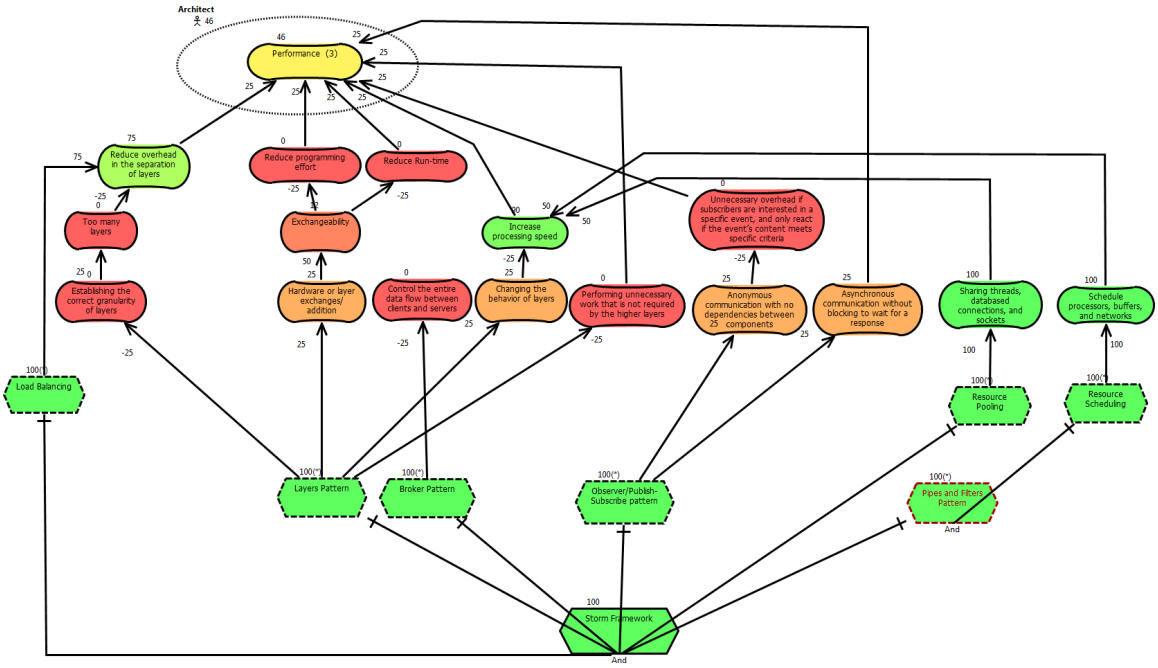


Figure 101 Evaluated GRL model of Storm in terms of Performance

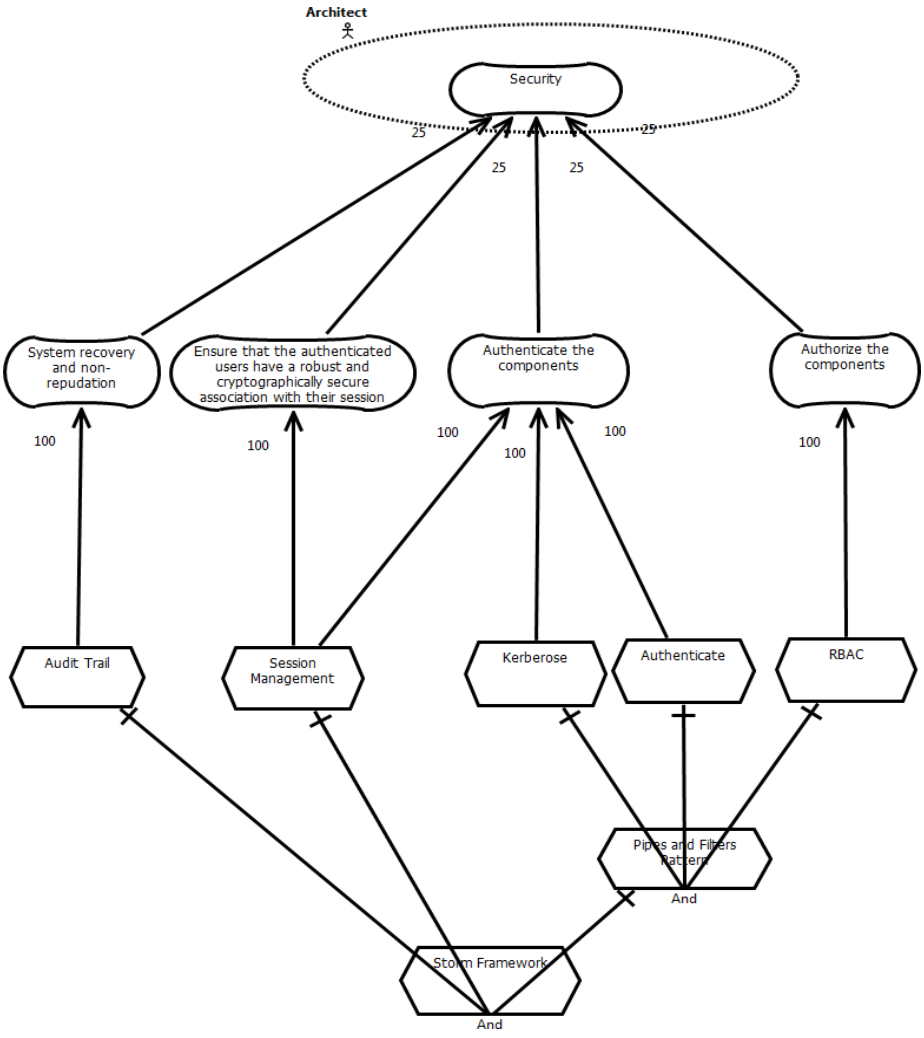


Figure 102 GRL model of Storm in terms of Security

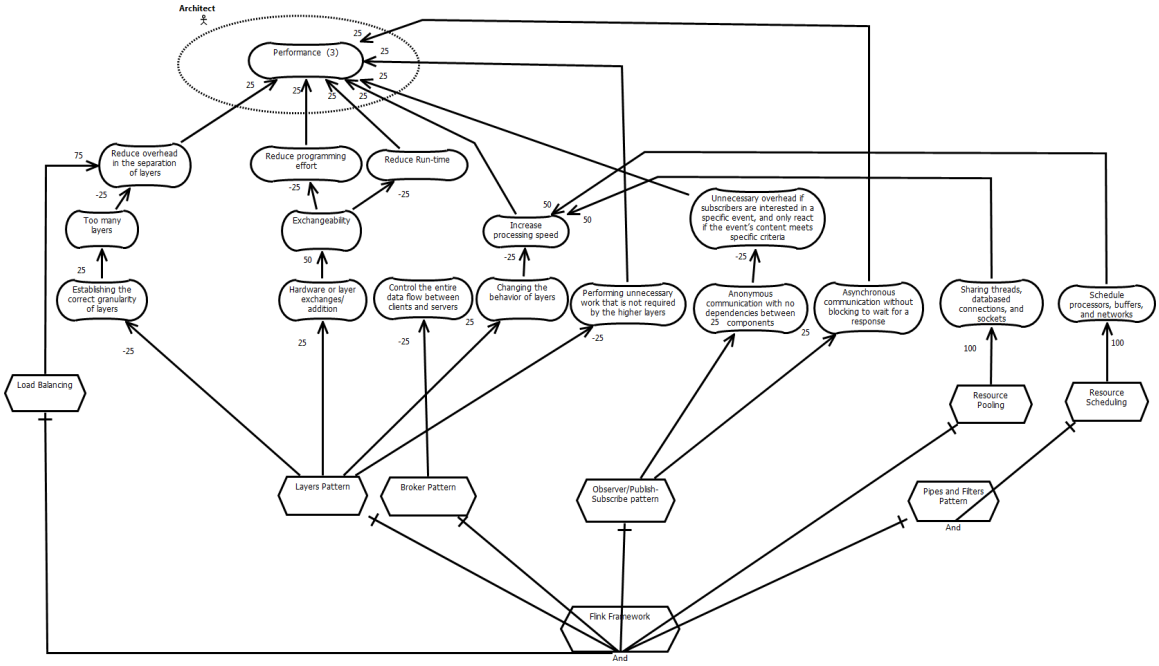


Figure 103 GRL model of Flink in terms of Performance

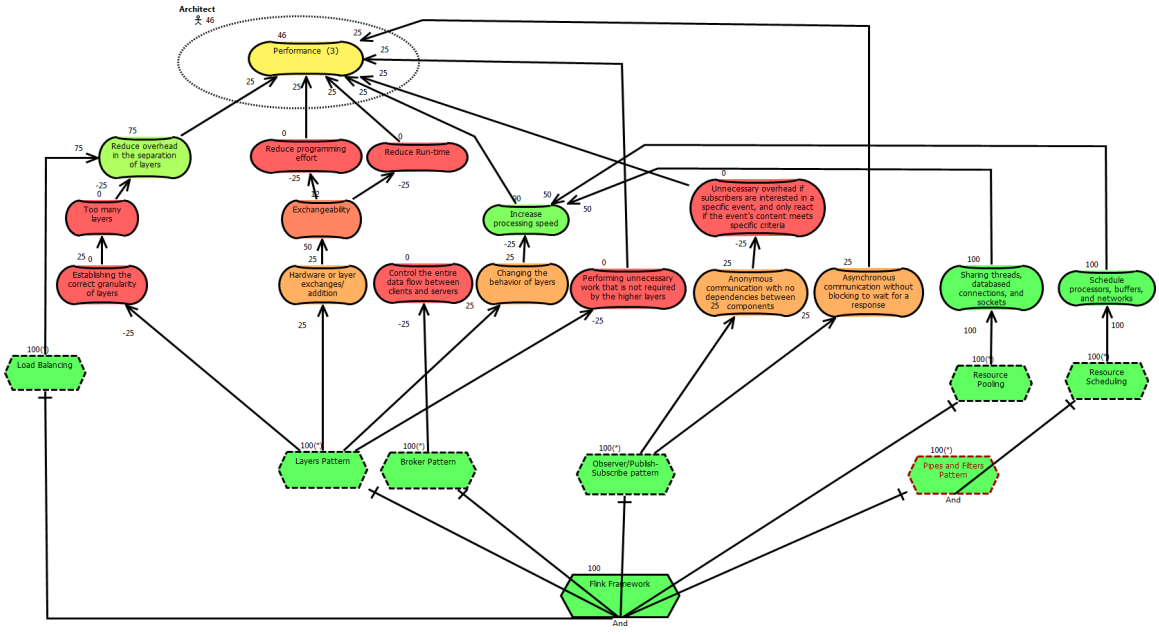


Figure 104 Evaluated GRL model of Flink in terms of Performance

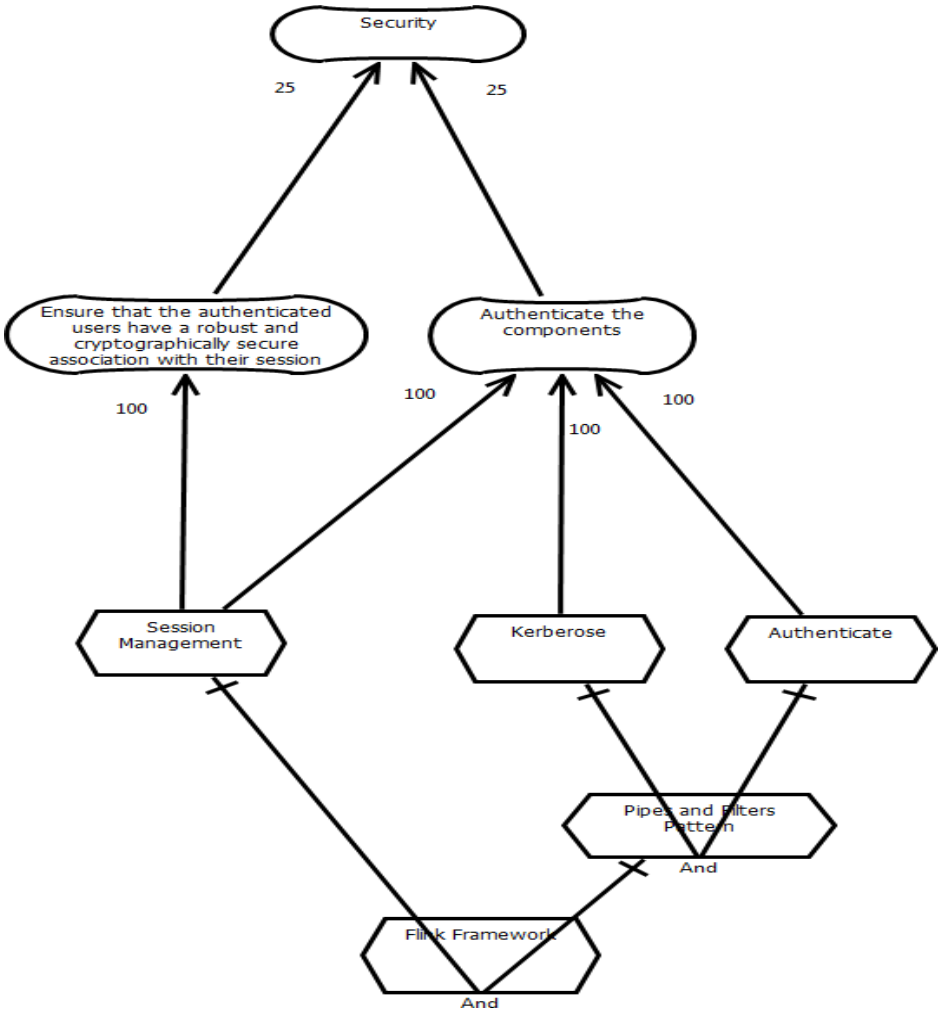


Figure 105 GRL model of Flink in terms of Security

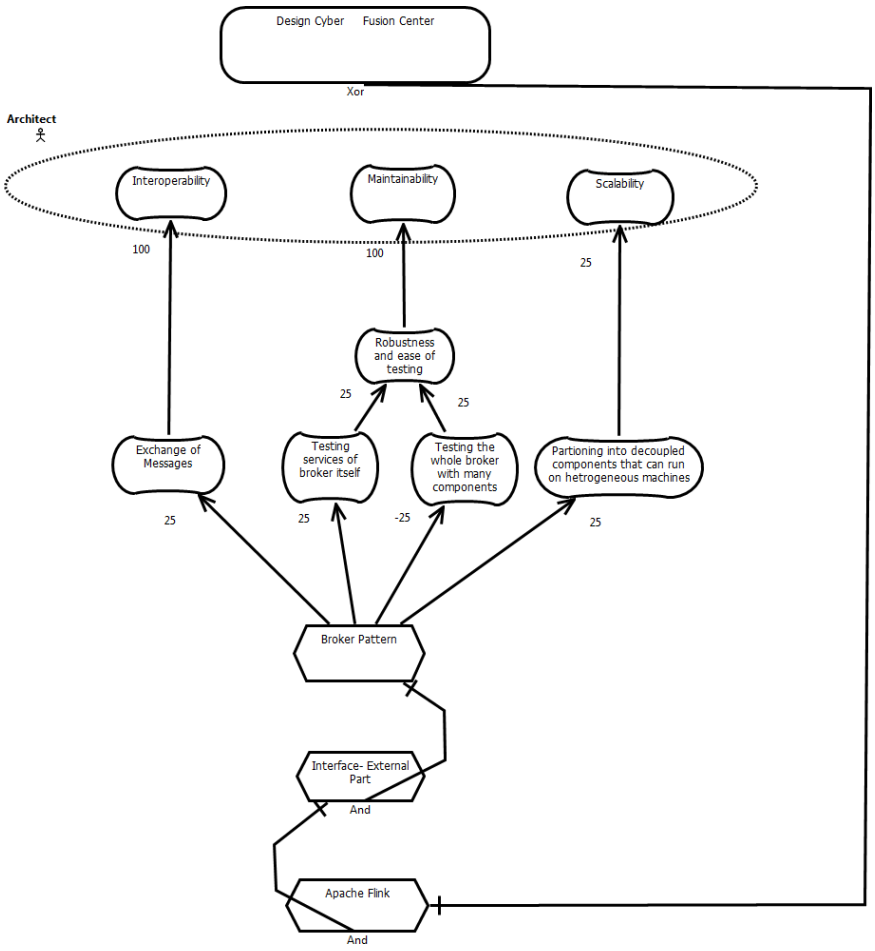


Figure 106 GRL model of Flink in terms of Interoperability, Maintainability, and Security

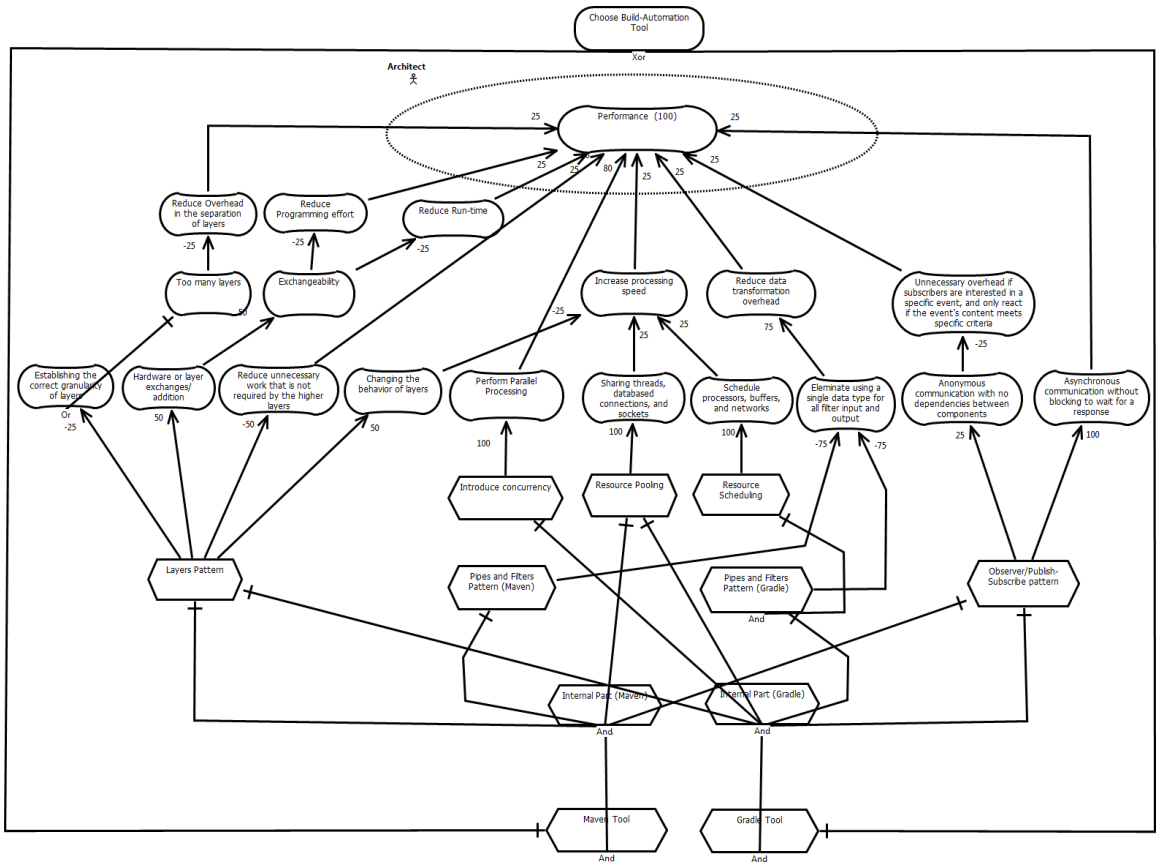
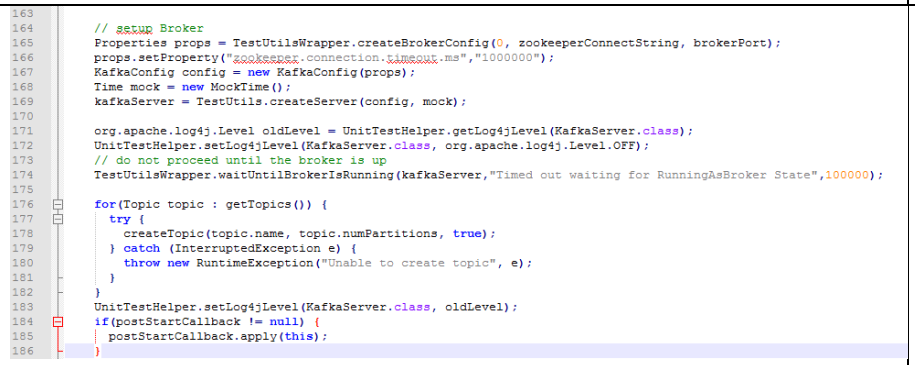


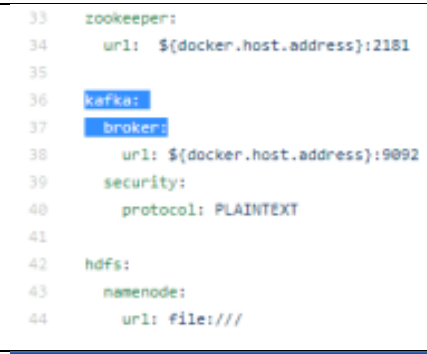
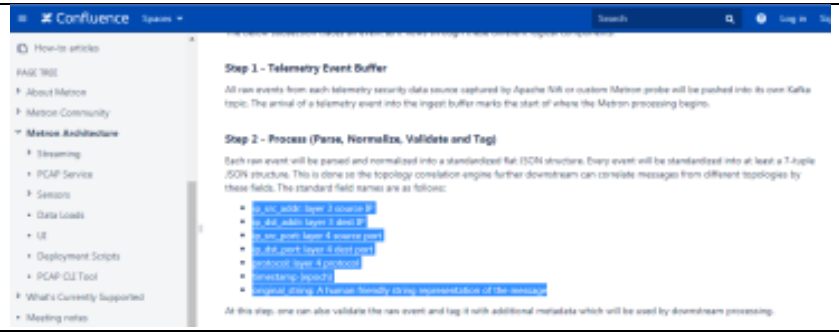
Figure 107 GRL model of Gradle and Maven in terms of Performance

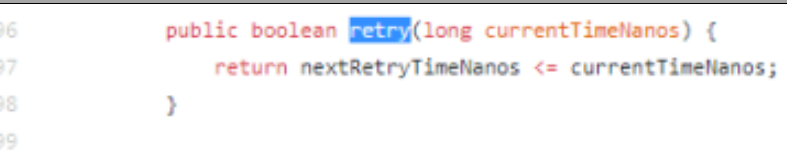
Appendix C: Validation Tables

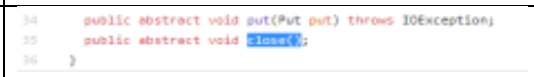
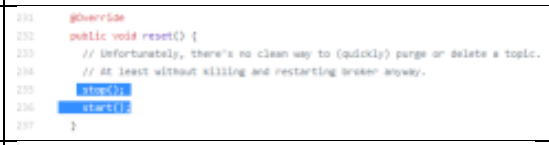
Table 84 Manual validation results for Storm

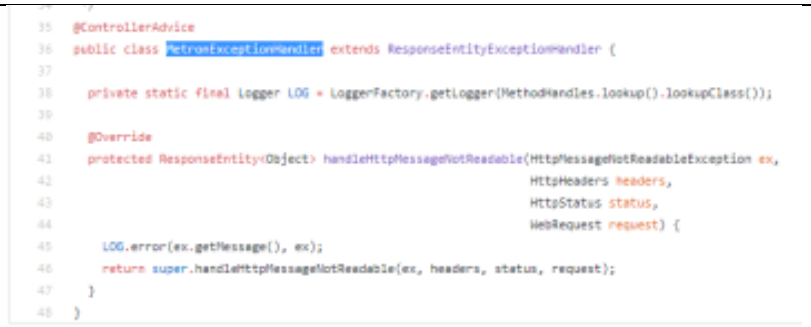
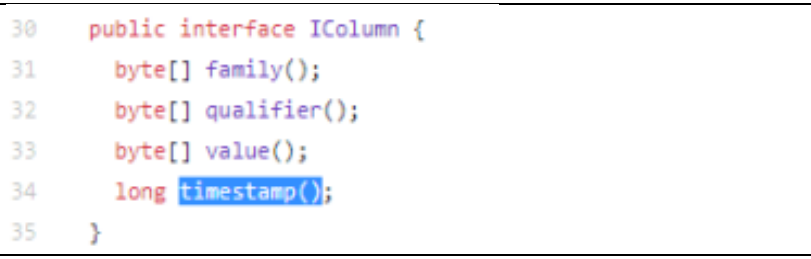
Patterns	Segment of Source Code	Package Name/Subsystem/Website
Observer/Publish-Subscribe [13]	 Metron framework is based on Kafka and in turn Kafka is based on the publish-subscribe model. The evidence is shown in following screen shot: 	Storm-Interface

		
Broker [13]		Storm-Platform Storm-Interface

		
Layers [13]		(https://cwiki.apache.org/confluence/display/STORM/Storm+Architecture).

The Related NFR	Tactics	Segment of Source Code	Package Name/Subsystem
Reliability/Availability	Retry [14]		KafkaSpoutRetryExponentialBackoff.java

		* The time stamp of the next retry is scheduled according to the exponential backoff formula (geometric progression): * nextRetry = failCount == 1 ? currentTime + initialDelay : currentTime + delayPeriod^(failCount-1) where failCount = 1, 2, 3, ... * nextRetry = Min(nextRetry, currentTime + maxDelay). * * By specifying a value for maxRetries lower than Integer.MAX_VALUE, the user decides to sacrifice guarantee of delivery for the previous * polled records in favor of processing more records. * * @param initialDelay initial delay of the first retry * @param delayPeriod the time interval that is the ratio of the exponential backoff formula (geometric progression) * @param maxRetries maximum number of times a tuple is retried before being acked and scheduled for commit * @param maxDelay maximum amount of time waiting before retrying * */	
	Close [14]		Storm-Interface
	Restart [14]		Storm-Platform
	Exception Handling [14]		Storm-Interface

			
	Time Stamp [14]		Storm-Interface

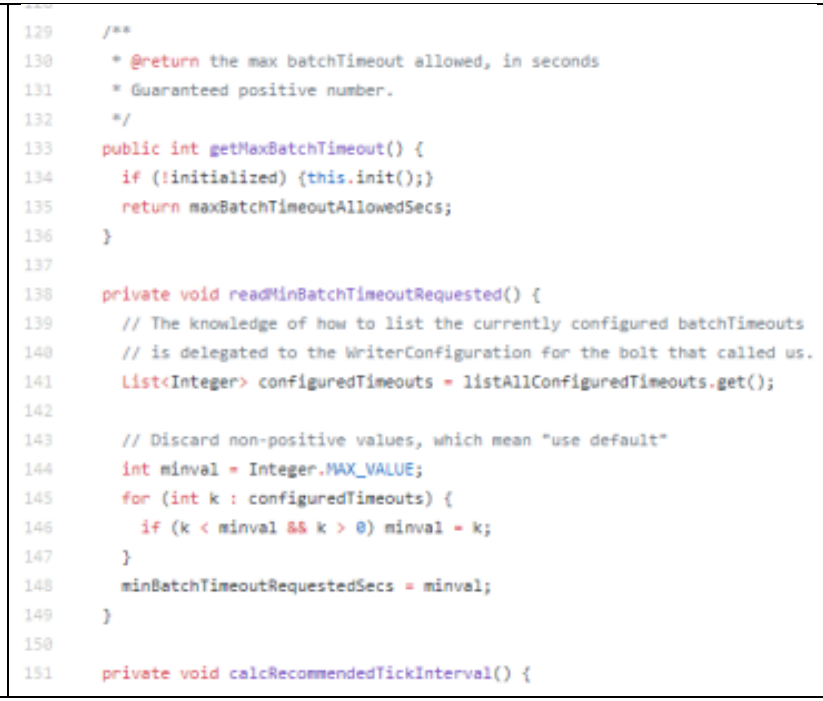
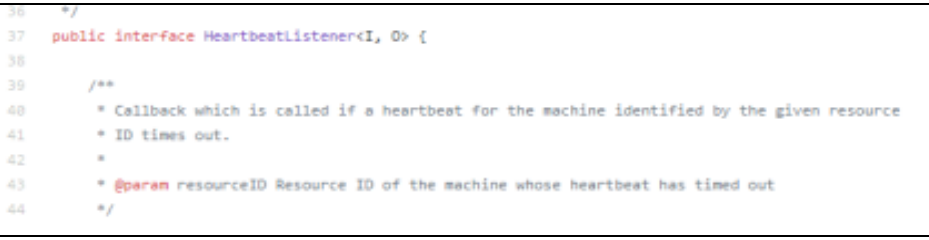
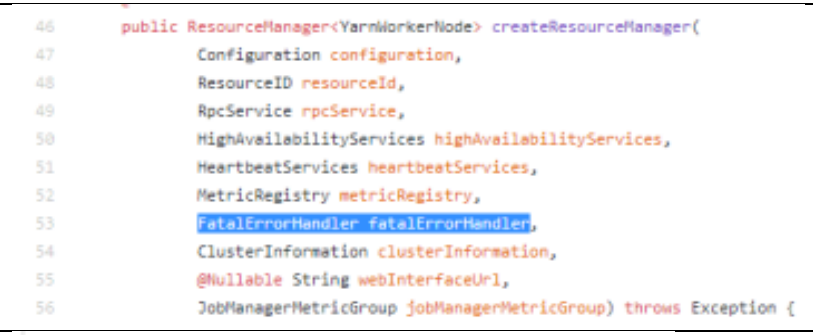
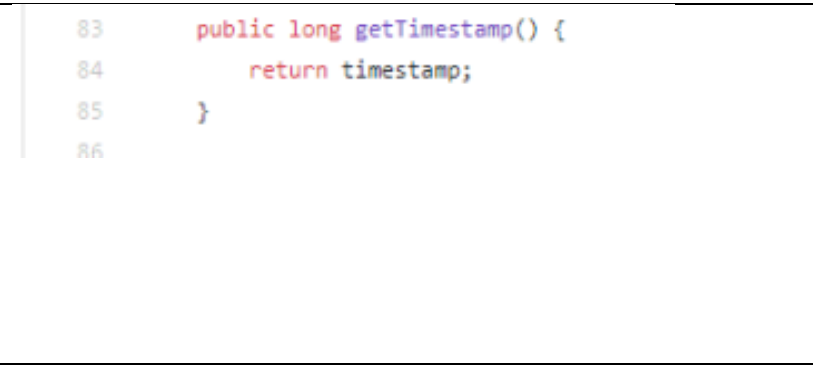
	Timeout [14]	 ``` /** * @return the max batchTimeout allowed, in seconds * Guaranteed positive number. */ public int getMaxBatchTimeout() { if (!initialized) {this.init();} return maxBatchTimeoutAllowedSecs; } private void readMinBatchTimeoutRequested() { // The knowledge of how to list the currently configured batchTimeouts // is delegated to the WriterConfiguration for the bolt that called us. List<Integer> configuredTimeouts = listAllConfiguredTimeouts.get(); // Discard non-positive values, which mean "use default" int minval = Integer.MAX_VALUE; for (int k : configuredTimeouts) { if (k < minval && k > 0) minval = k; } minBatchTimeoutRequestedSecs = minval; } private void calcRecommendedTickInterval() { ```	Storm-Platform

Table 85 Manual validation results for Flink

Patterns	Segment of Source Code	Package Name/Subsystem
Layers [13]		Flink-Libraries
Observer/Publish-Subscribe [13]		Flink-Runtime

Pipes and Filters [13]	 94 95 /** 96 * A thin wrapper layer over {@link DataStream#filter(FilterFunction)}. 97 * 98 * @param filter The FilterFunction that is called for each element of the DataStream. 99 * @return The filtered {@link PythonDataStream}. 100 */ 101 public PythonSingleOutputStreamOperator filter(FilterFunction<PyObject> filter) throws IOException { 102 return new PythonSingleOutputStreamOperator(stream.filter(new PythonFilterFunction(filter))); 103 } 104 105 /** 106 * A thin wrapper layer over {@link DataStream#map(MapFunction)}. 107 *	Flink-Libraries

The Related NFR	Tactics	Segment of Source Code	Package Name/Subsystem/Website
Reliability/Availability	Close [14]	 222 223 // -- 224 // shutdown 225 // -- 226 227 @Override 228 public void close() throws Exception { 229 if (enterUnlessClosed()) { 230 try { 231 try { 232 // this class' own cleanup logic 233 resourceManagerLeaderElectionService.shutdown(); 234 dispatcher.shutdownNow(); 235 } 236 finally { 237 // in any case must we call the parent cleanup logic 238 super.close(); 239 } 240 } 241 finally { 242 exit(); 243 } 244 } 245 } 246 }	Flink-Yarn

	Restart [14]		Flink-Runtime
	Exception Handling [14]		Flink-Yarn
	Time Stamp [14]		Flink-Connector

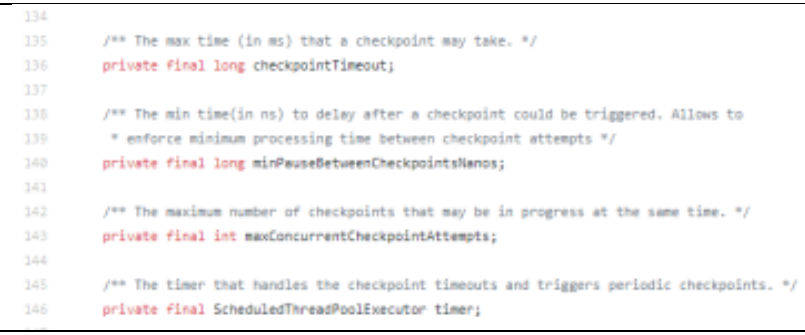
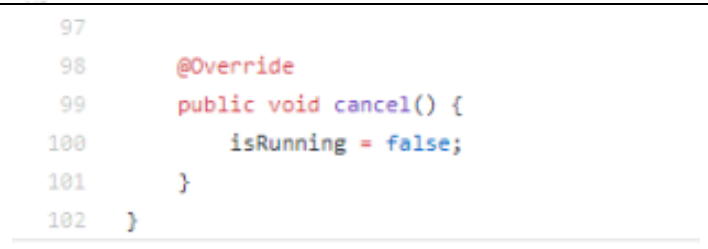
			Flink-Streaming
	Timeout [14]		Flink-Runtime
	Cancel [14]		Flink-Contrib/Flink-Connector

Table 86 Manual validation results for Spark

Tactic/Pattern	Source file	Detected Terms	Code Where Terms are Detected	Description	Compared with Our Manual Search
Kerberos	Client.java	Kerbero, credent, princip	if (UserGroupInformation.isSecurityEnabled()) { // Note: Credentials class is marked as LimitedPrivate for HDFS and MapReduce Credentials credentials = new Credentials(); String tokenRenewer = conf.get(YarnConfiguration.RM_PRINCIPAL); if (tokenRenewer == null || tokenRenewer.length() == 0) { throw new IOException("Can't get Master Kerberos principal for the RM to use as renewer"); }	All of the codes in the previous column, which are related to the Kerberos tactic give us an indication that Kerberos tactic is applied in the Metron framework. This is clear in the codes when Kerberos tries to authenticate the users using username, and key tab.	Detected by our manual search and Archie
	MetronResetConstants.java	Kerbero, princip	public static final String KERBEROS_ENABLED_SPRING_PROPERTY = "kerberos.enabled"; public static final String KERBEROS_PRINCIPLE_SPRING_PROPERTY = "kerberos.principal"; public static final String KERBEROS_KEYTAB_SPRING_PROPERTY = "kerberos.keytab";		
	HadoopConfig.java	Kerbero, princip	if (environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)) { UserGroupInformation.setConfiguration(configuration); String keyTabLocation = environment.getProperty(MetronRestConstants.KERBEROS_KEYTAB_SPRING_PROPERTY); String userPrincipal = environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPERTY); UserGroupInformation.loginUserFromKeytab(userPrincipal, keyTabLocation); }		

	IndexConfig.java	Kerbero	config.setKerberosEnabled(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false));		
	KafkaConfig.java	Kerbero	if (environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)) { props.put("security.protocol", KafkaUtils.INSTANCE.normalizeProtocol(environment.getProperty(MetronRestConstants.KAFKA_SECURITY_PROTOCOL_SPRING_PROPERTY))); }		
	ResetTemplateConfig.java	Kerbero, princip	import org.springframework.security.kerberos.client.KerberosRestTemplate; public RestTemplate restTemplate() { if (environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)) { String keyTabLocation = environment.getProperty(MetronRestConstants.KERBEROS_KEYTAB_SPRING_PROPERTY); String userPrincipal = environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPERTY); return new KerberosRestTemplate(keyTabLocation, userPrincipal); } else { return new RestTemplate(); } }		

	HadoopConfigTest.java	Kerbero, authent, princip	public void configurationShouldReturnProperKerberosConfiguration() throws IOException { when(environment.getProperty(MetronRestConstants.KERBEROS_KEYTAB_SPRING_PROPERTY)).thenReturn("metron keytabLocation"); when(environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPE RTY)).thenReturn("metron principal"); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(true); public void configurationShouldReturnProperConfiguration() throws IOException { when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(false); assertEquals("simple", configuration.get("hadoop.security.authentication")); when(environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPE RTY)).thenReturn("metron principal"); UserGroupInformation.loginUserFromKeytab("metron keytabLocation", "metron principal");		
	KafkaConfigTest.java	Kerbero	when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(false); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(true); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(false); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPER TY, Boolean.class, false)).thenReturn(true);		

	RestTemplateConfigTest.java	Kerbero, princip	import org.springframework.security.kerberos.client.KerberosRestTemplate; @PrepareForTest({RestTemplateConfig.class, KerberosRestTemplate.class, RestTemplate.class}) when(environment.getProperty(MetronRestConstants.KERBEROS_KEYTAB_SPRING_PROPERTY)).thenReturn("metron keytabLocation"); when(environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPERTY)).thenReturn("metron principal"); whenNew(KerberosRestTemplate.class).withParameterTypes(String.class, String.class).withArguments("metron keytabLocation", "metron principal").thenReturn(null); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)).thenReturn(true); verifyNew(KerberosRestTemplate.class).withArguments("metron keytabLocation", "metron principal"); when(environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)).thenReturn(false);		
	KafkaServiceImpl.java	Kerbero, authent, princip	if (environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)){ User principal = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal(); String user = principal.getUsername(); cmd.add("--allow-principal"); String user = principal.getUsername();		

	StormCLIWrapper.java StormCLIWrapperTest.java	Kerbero, princip	boolean kerberosEnabled = environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false); if (kerberosEnabled && topologyOptionsDefined) { command.add("-e"); command.add(environment.getProperty(MetronRestConstants.PARSER_TOPOLOGY_OPTIONS_SPRING_PROPERTY)); } if (environment.getProperty(MetronRestConstants.KERBEROS_ENABLED_SPRING_PROPERTY, Boolean.class, false)) { String keyTabLocation = environment.getProperty(MetronRestConstants.KERBEROS_KEYTAB_SPRING_PROPERTY); String userPrincipal = environment.getProperty(MetronRestConstants.KERBEROS_PRINCIPLE_SPRING_PROPERTY); String[] kinitCommand = {"kinit", "-kt", keyTabLocation, userPrincipal}; runCommand(kinitCommand); }		
	ParserToplogyCLITest.java	Kerbero	config.setSecurityProtocol("KERBEROS"); input -> input.getSecurityProtocol().equals("KERBEROS") config2.setSecurityProtocol("KERBEROS"); config2.setSecurityProtocol("KERBEROS"); input -> input.getSecurityProtocol().equals("KERBEROS");		

	AccessConfig.java	Kerbero	private Boolean isKerberosEnabled = false; * @return True if clients should be configured for Kerberos public Boolean getKerberosEnabled() { return isKerberosEnabled; } public void setKerberosEnabled(Boolean kerberosEnabled) { isKerberosEnabled = kerberosEnabled; }		

Heartbeat	ApplicationMaster.java	Heartbeat, beat	The <code>ApplicationMaster</code> needs to send a heartbeat to the * <code>ResourceManager</code> at regular intervals to inform the * <code>ResourceManager</code> that it is up and alive. The * {@link ApplicationMasterProtocol#allocate} to the <code>ResourceManager</code> from the * <code>ApplicationMaster</code> acts as a heartbeat. // This will start heartbeating to the RM	It is very clear in the file of ApplicationMaster.java that Metron has applied both Heartbeat and Ping/Echo tactics. This is clear in the documented comments of ApplicationMaster.java file. So, there is an echo component (ApplilcationMaster), which sends the request and ping component (ResourceManager) which receives the request from the binging component. The applicationMaster also determines that ApplicationMaster is alive and responding correctly.	Detected by our manual search and Archie

	ContainerRequestListener.java	Heartbeat, beat	@Override public float getProgress() { // set progress to deliver to RM on next heartbeat float progress = 0; return progress; }	Another evidence here has the same description above.	
Ping/Echo	ElasticsearchUtils.java	ping	settingsBuilder.put("cluster.name", globalConfiguration.get("es.clustername")); settingsBuilder.put("client.transport.ping_timeout", esSettings.getOrDefault("client.transport.ping_timeout","500s")); setXPackSecurityOrNone(settingsBuilder, esSettings);	Ping_timeout can gives us an indication that Ping/Echo is there. As I know in Ping/Echo tactic the pinging component needs threshold to tell it how long to wait for the echo before considering the pinged component to have failed ("time out"). This is clear in the code when they used ping-timeout and set it to be 500s as a waiting time.	Detected by our manual search and Archie

	MaasIntegrationTest.java	echo	if (endpoints != null && endpoints.size() == 1) { LOG.trace("Found endpoints: " + endpoints.get(0)); String output = makeRESTcall(new URL(endpoints.get(0).getEndpoint().getUrl() + "/echo/casey")); if (output.contains("casey")) { passed = true; break; } }	The keyword echo just discovered by Archie tool but it doesn't show anything related to the Ping/Echo tactic. It just echo word used for text printing. So, I can ignore this evidence.	
	HDFSDDataPruner.java	echo	public static void main(String... argv) throws IOException, java.text.ParseException, ClassNotFoundException, InterruptedException { /** * Example * start=$(date -d '30 days ago' +%m/%d/%Y) * yarn jar Metron-DataLoads-0.1BETA.jar org.apache.metron.dataloads.bulk.HDFSDataPruner -f hdfs://ec2-52-36-25-217.us-west-2.compute.amazonaws.com:8020 -g '/apps/metron/enrichment/indexed/bro_doc/*enrichment-*' -s $(date -d '30 days ago' +%m/%d/%Y) -n 1; * echo ${start} **/	The same description as above.	
	CIFHbaseAdapter.java	ping	@SuppressWarnings({ "rawtypes", "deprecation" }) protected Map getCIFObject(String key) { LOGGER.debug("=======Pinging HBase For: {}", key);	Pinging keyword here just describes the Hbase. However, it can't be considered as an evidence of using Bing/Echo tactic.	
	AlertsUIServiceImpl.java	layer	* The default service layer implementation of {@link AlertsUIService}.		

Layers pattern			* * @see AlertsUIService */	The comments show that there are layers called service layer and link layer. This means that Metron has different layers in its architecture. Consequently, I can say that Metron applies the Layered pattern.	Detected by our manual search and Archie.
	KafkaServiceImpl.java	layer	* The default service layer implementation of {@link KafkaService}. * * @see KafkaService */ @Service		
	PcapHelper.java	layer	public static byte[] getPcapGlobalHeader(Endianness endianness) { if(swapBytes(endianness)) { //swap return new byte[] { (byte) 0xd4, (byte) 0xc3, (byte) 0xb2, (byte) 0xa1 //swapped magic number 0xa1b2c3d4 , 0x02, 0x00 //swapped major version 2 , 0x04, 0x00 //swapped minor version 4 , 0x00, 0x00, 0x00, 0x00 //GMT to local tz offset (= 0) , 0x00, 0x00, 0x00, 0x00 //sigfigs (= 0) , (byte) 0xff, (byte) 0xff, 0x00, 0x00 //snaplen (=65535) , 0x01, 0x00, 0x00, 0x00 // swapped link layer header type (1 = ethernet) }; } else { //no need to swap return new byte[] { (byte) 0xa1, (byte) 0xb2, (byte) 0xc3, (byte) 0xd4 //magic number 0xa1b2c3d4 , 0x00, 0x02 //major version 2 , 0x00, 0x04 //minor version 4 , 0x00, 0x00, 0x00, 0x00 //GMT to local tz offset (= 0) , 0x00, 0x00, 0x00, 0x00 //sigfigs (= 0) , 0x00, 0x00, (byte) 0xff, (byte) 0xff //snaplen (=65535) , 0x00, 0x00, 0x00, 0x01 // link layer header type (1 = ethernet) }; } }		
	SimpleStormKafkaBuilder.java	layer	** * This is a convenience layer on top of the KafkaSpoutConfig.Builder available in storm-kafka-client.		

			* The justification for this class is two-fold. First, there are a lot of moving parts and a simplified * approach to constructing spouts is useful. Secondly, and perhaps more importantly, the Builder pattern * is decidedly unfriendly to use inside of Flux. Finally, I can make things a bit more friendly by only requiring * zookeeper and automatically figuring out the brokers for the bootstrap server. * * @param <K> The kafka key type * @param <V> The kafka value type */		

Exception Handling tactic	MaaSFunctions.java	Exception, error, handl, catch, try	import java.net.MalformedURLException; import org.apache.metron.stellar.dsl.ParseException; public Object apply(List<Object> args, Context context) throws ParseException { if(args.size() < 2) { throw new ParseException("Unable to execute model_apply. " + "Expected arguments: endpoint_map:map, " + " [endpoint method:string], model_args:map"); } catch (Exception e) { LOG.error(e.getMessage(), e); if (discoverer != null) { try { URL u = new URL(modelUrl); discoverer.blacklist(u); } catch (MalformedURLException e1) { throw new IllegalStateException("Unable to initialize function: Cannot find zookeeper client."); } catch(Exception ex) { LOG.error(ex.getMessage(), ex); import java.lang.invoke.MethodHandles; protected static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().look	It is very clear that Exception Handling tactic is there because of the catch-try statements. This means there is a tactic is used to catch and handle the errors and the faults.	Detected by our manual search and Archie

	AlertsUIControllerIntegrationTest.java	Exception, catch, try	try { runner.start(); } catch (UnableToStartException e) { e.printStackTrace(); } }		

	EnrichmentJoinBolt.java	Exception, handl, try	if(sourceType == null) { String errorMessage = "Unable to find source type for message: " + message; throw new IllegalStateException(errorMessage); } Map<String, ConfigHandler> handlerMap = getFieldToHandlerMap(sourceType); for (String enrichmentType : fieldMap.keySet()) { ConfigHandler handler = handlerMap.get(enrichmentType); List<String> subgroups = handler.getType().getSubgroups(handler.getType().toConfig(handler.getConfig())); for(String subgroup : subgroups) { streamIds.add(Joiner.on(":").join(enrichmentType, subgroup)); } protected Map<String, ConfigHandler> getFieldToHandlerMap(String sensorType) { else { LOG.error("Trying to retrieve a field map with sensor type of null"); }		
	EnrichmentSplitterBolt.java	Exception, try, catch, handl,	try { message = (JSONObject) parser.parse(new String(data, "UTF8")); message.put(getClass().getSimpleName().toLowerCase() + ".splitter.begin.ts", "" + System.currentTimeMillis());		

			} catch (ParseException | UnsupportedEncodingException e) { e.printStackTrace(); } Map<String, ConfigHandler> fieldToHandler = getFieldToHandlerMap(sensorType); enrichmentTypes.addAll(fieldToHandler.keySet()); ConfigHandler retriever = fieldToHandler.get(enrichmentType);		
	ThreatIntelJoinBolt.java	Handl, error	import java.lang.invoke.MethodHandles; import org.apache.metron.common.configuration.enrichment.handler.ConfigHandl protected Map<String, ConfigHandler> getFieldToHandlerMap(String sensorType) { else { LOG.error("Trying to retrieve a field map with sensor type of null");		
	ThreatIntelSplitterBolt.java	handl	import org.apache.metron.common.configuration.enrichment.handler.ConfigHandl protected Map<String, ConfigHandler> getFieldToHandlerMap(String sensorType) {		
	IndexDaoFactory.java	Excepti on, try	public static IndexDao combine(Iterable<IndexDao> daos, Function<IndexDao, IndexDao> daoTransformation) throws ClassNotFoundException, NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException { If(numDaos == 0) { throw new IllegalArgumentException("Trying to combine 0 dao's into a DAO is not a supported configuration."); } if(numDaos == 1) { return daoTransformation.apply(Iterables.getFirst(daos, null)); }		

	LongLiteralEvaluator.java	Exception, try	if (value.endsWith("l") || value.endsWith("L")) { value = value.substring(0, value.length() - 1); // Drop the 'L' or 'l'. Long.parseLong does not accept a string with either of these. return new Token<>(Long.parseLong(value), Long.class, contextVariety); } else { // Technically this should never happen, but just being safe. throw new ParseException("Invalid format for long. Failed trying to parse a long with the following value: " + value); } } public Token<Long> evaluate(StellarParser.LongLiteralContext context, FrameContext.Context contextVariety) { if (context == null) { throw new IllegalArgumentException("Cannot evaluate a context that is null."); }		
	LongLiteralEvaluatorTest.java	Exception, try	public void verifyHappyPathEvaluation() throws Exception { when(context.getText()).thenReturn("100L"); public void verifyNumberFormationExceptionWithEmptyString() throws Exception { exception.expect(ParseException.class); exception.expectMessage("Invalid format for long. Failed trying to parse a long with the following value: "); public void throwIllegalArgumentExceptionWhenContextIsNull() throws Exception { exception.expect(IllegalArgumentException.class); exception.expectMessage("Cannot evaluate a context that is null.");		

Authenticate		authent	 @Injectable() export class AuthenticationService { private static USER_NOT_VERIFIED = 'USER-NOT-VERIFIED'; private currentUser: string = AuthenticationService.USER_NOT_VERIFIED; userUrl = this.appConfigService.getApiRoot() + '/user'; logoutUrl = this.appConfigService.getApiRoot() + '/logout'; onLoginEvent: ReplaySubject<boolean> = new ReplaySubject<boolean>(); $onLoginEvent = this.onLoginEvent.asObservable(); constructor(private http: HttpClient, private router: Router, private globalConfigService: GlobalConfigService, private dataSource: DataSource, private appConfigService: AppConfigService) { this.init(); }	It seems from the code that Authenticate tactic is really applied in the Metron framework. This is because the code tries to get all user's details, username, and the context to authenticate the user. Consequently, making everything very secured. AuthenticationService class is also another evidence.	Detected by our manual search and Archie
	AlertsUIServiceImpTest.java	authent	SecurityContextHolder.getContext().setAuthentication(authentication); Authentication authentication = Mockito.mock(Authentication.class); UserDetails userDetails = Mockito.mock(UserDetails.class); when(authentication.getPrincipal()).thenReturn(userDetails); when(userDetails.getUsername()).thenReturn(user1); SecurityContextHolder.getContext().setAuthentication(authentication); }		
	HadoopConfigTest.java	authent	assertEquals("simple", configuration.get("hadoop.security.authentication"));		
Broker pattern	KafkaConfig.java StormCLIWrapper.java	broker	props.put("bootstrap.servers", environment.getProperty(MetronRestConstants.KAFKA_BROKER_URL_SPRING_PROPERTY)); // kafka broker command.add(environment.getProperty(MetronRestConstants.KAFKA_BROKER_URL_SPRING_PROPERTY));	It is very clear that the Broker pattern is applied here. Kafka is already built based on using Broker pattern. So, broker element is there as shown in the code.	Detected by our manual search and Archie
Time Stamp	ElasticsearchDao.java	timestamp	@Override public void patch(RetrieveLatestDao retrieveLatestDao, PatchRequest request, Optional<Long> timestamp) throws OriginalNotFoundException, IOException { updateDao.patch(retrieveLatestDao, request, timestamp); } @Override	It seems that Time Stamp tactic is used here because all the codes in the previous column are talking about detecting the incorrect sequences of events using what is called timestamp.	Detected by our manual search and Archie

	ElasticsearchMetaAlretDao.java		public void replace(ReplaceRequest request, Optional<Long> timestamp) throws IOException { updateDao.replace(request, timestamp); } public void patch(RetrieveLatestDao retrieveLatestDao, PatchRequest request, Optional<Long> timestamp) throws OriginalNotFoundException, IOException { metaAlertUpdateDao.patch(retrieveLatestDao, request, timestamp); }		
	ElasticsearchWriter.java	timestamp	Object ts = esDoc.get("timestamp"); if(ts != null) { indexRequestBuilder = indexRequestBuilder.setTimestamp(ts.toString()); } bulkRequest.add(indexRequestBuilder); }		
	PersistentAccessTracker.java	timestamp	public void persist(boolean force) { synchronized(sync) { if(force || (System.currentTimeMillis() - timestamp) >= maxMillisecondsBetweenPersists) { //persist try { AccessTrackerUtil.INSTANCE.persistTracker(accessTrackerTable, accessTrackerColumnFamily, new AccessTrackerKey(name, containerName, timestamp), underlyingTracker); timestamp = System.currentTimeMillis(); reset(); } catch (IOException e) { LOG.error("Unable to persist access tracker.", e); } } } }		

Resource Pooling	PcapFinalizer.java	Pool, thread, rocessor	ForkJoinPool tp = new ForkJoinPool(parallelism); private static int getNumThreads(String numThreads) throws JobException { String numThreadsStr = ((String) numThreads).trim().toUpperCase(); try { if (numThreadsStr.endsWith("C")) { Integer factor = Integer.parseInt(numThreadsStr.replace("C", "")); return factor * Runtime.getRuntime().availableProcessors(); } else { return Integer.parseInt(numThreadsStr); } } catch (NumberFormatException e) { throw new JobException(format("Unable to set number of threads for finalizing from property value '%s'", numThreads)); }	It see seems that there is a thread pool is used to store a set of threads. Consequently, I can say Resource Pooling tactic is applied in Metron.	Detected by our manual search and Archie
	UnifiedEnrichmentBolt.java	Pool, thread, param	/** * The number of threads in the threadpool. One threadpool is created per process. * This is a topology-level configuration */ public static final String THREADPOOL_NUM_THREADS_TOPOLOGY_CONF = "metron.threadpool.size"; /** * The type of threadpool to create. This is a topology-level configuration. */ public static final String THREADPOOL_TYPE_TOPOLOGY_CONF = "metron.threadpool.type"; /** * Figure out how many threads to use in the thread pool. The user can pass an arbitrary object, so parse it * according to some rules. If it's a number, then cast to an int. IF it's a string and ends with "C", then strip * the C and treat it as an integral multiple of the number of cores. If it's a string and does not end with a C, then treat * it as a number in string form. * @param numThreads	This is another evidence of having a shared pool of set of threads.	

```
 * @return
 */
private static int getNumThreads(Object numThreads) {
  if(numThreads instanceof Number) {
    return ((Number)numThreads).intValue();
  }
  else if(numThreads instanceof String) {
    String numThreadsStr = ((String)numThreads).trim().toUpperCase();
    if(numThreadsStr.endsWith("C")) {
      Integer factor = Integer.parseInt(numThreadsStr.replace("C", ""));
      return factor*Runtime.getRuntime().availableProcessors();
    }
    else {
      return Integer.parseInt(numThreadsStr);
    }
  }
  return 2*Runtime.getRuntime().availableProcessors();
}

WorkerPoolStrategies workerPoolStrategy = WorkerPoolStrategies.FIXED;
  if(map.containsKey(THREADPOOL_TYPE_TOPOLOGY_CONF)) {
    workerPoolStrategy =
WorkerPoolStrategies.valueOf(map.get(THREADPOOL_TYPE_TOPOLOGY_CO
NF) + "");
  }
if(map.containsKey(THREADPOOL_NUM_THREADS_TOPOLOGY_CONF)) {
    int numThreads =
getNumThreads(map.get(THREADPOOL_NUM_THREADS_TOPOLOGY_CONF))
;
    ConcurrencyContext.get(strategy).initialize(numThreads, maxCacheSize,
maxTimeRetain, workerPoolStrategy, LOG, captureCacheStats);
  }
else {
    throw new IllegalStateException("You must pass " +
THREADPOOL_NUM_THREADS_TOPOLOGY_CONF + " via storm config.");
  }
```

Audit Taril	GrokWebSphereParserTest .java	Audit, log, string	String testString = "<134>Apr 15 17:17:34 SAGPXMLQA333 [0x8240001c][audit][info] trans 191) admindefaultsystem*): " + "ntp-service 'NTP Service' - Operational state down:"; assertEquals("audit", parsedJSON.get("event_type")); assertEquals("login", parsedJSON.get("event_subtype")); assertEquals("rick007", parsedJSON.get("username")); assertEquals("120.43.200.6", parsedJSON.get("ip_src_addr")); } @Test public void tetsParseLogoutLine() throws Exception { //Set up parser, parse message GrokWebSphereParser parser = new GrokWebSphereParser(); parser.configure(parserConfig); String testString = "<134>Apr 15 18:02:27 PHIXML3RWD [0x81000019][auth][info] [14.122.2.201]: " + "User 'hjpotter' logged out from 'default'."; List<JSONObject> result = parser.parse(testString.getBytes()); JSONObject parsedJSON = result.get(0);	It seems that the code is doing auditing process (detecting attackers). This is clear in the following statements where it checks the type of event, the logged username, and the IP address: assertEquals("audit", parsedJSON.get("event_type")); assertEquals("rick007", parsedJSON.get("username")); assertEquals("120.43.200.6", parsedJSON.get("ip_src_addr"));	Detected by our manual search and Archie
	PcapByteInputStream.java PcapByteOuputStream.java PcapMerger.java PcapHelper.java	audit	@Override public void close() throws IOException { is.close(); // $codepro.audit.disable closeInFinally } public PcapByteInputStream(byte[] pcap, int offset, int length) throws IOException { is = new DataInputStream(new ByteArrayInputStream(pcap, offset, length)); // $codepro.audit.disable // closeWhereCreated readGlobalHeader(); }	It doesn't seem that there is an audit trail tactic implemented in the codes of all these classes.	

			public static void merge(ByteArrayOutputStream baos, byte[]... pcaps) // $codepro.audit.disable // overloadedMethods throws IOException { merge(baos, Arrays.asList(pcaps)); }		
PBAC	HdfsWriter.java	permiss	See the License for the specific language governing permissions and * limitations under the License. public class HdfsWriter implements BulkMessageWriter<JSONObject>, Serializable { List<RotationAction> rotationActions = new ArrayList<>(); FileRotationPolicy rotationPolicy = new NoRotationPolicy(); SyncPolicy syncPolicy; FileNameFormat fileNameFormat; Map<SourceHandlerKey, SourceHandler> sourceHandlerMap = new HashMap<>(); int maxOpenFiles = 500; transient StellarProcessor stellarProcessor; transient Map stormConfig; transient SyncPolicyCreator syncPolicyCreator;	It doesn't seem that HdfsWriter class relates to PBAC tactic.	Detected by Archie but not by our manual search
	HDFSIndexingIntegrationTest.java	Oc, permiss	@Override public ProcessorResult<List<Map<String, Object>>> getResult() { ProcessorResult.Builder<List<Map<String,Object>>> builder = new ProcessorResult.Builder(); return builder.withResult(docs).withProcessErrors(errors).build(); } See the License for the specific language governing permissions and * limitations under the License. */	It does not seem that the existence of oc and permiss in the HDFSIndexingIntegrationTest.java can determine that there is a PBAC tactic.	

RBAC	Client.java	Access, control, secur, permiss	fs.setPermission(dst, new FsPermission((short)0755)); FileStatus scFileStatus = fs.getFileStatus(dst); LocalResource scRsrc = LocalResource.newInstance(ConverterUtils.getYarnUrlFromURI(dst.toUri()), LocalResourceType.FILE, LocalResourceVisibility.APPLICATION, scFileStatus.getLen(), scFileStatus.getModificationTime()); localResources.put(fileDstPath, scRsrc); return dst; } // Timeline domain reader access control private String viewACLs = null; // Timeline domain writer access control private String modifyACLs = null; import org.apache.hadoop.security.token.Token; import org.apache.hadoop.fs.permission.FsPermission; fs.setPermission(dst, new FsPermission((short)0755));	It is clear that RBAC is applied in Client.java. this is clear in the code especially in the following statement: LocalResourceType.FILE, LocalResourceVisibility.APPLICATION, scFileStatus.getLen(), scFileStatus.getModificationTime()); There is an authorization process to ensure that the client has access on the file data such as: file status, length, the modification,.etc.	Detected by our manual search and Archie
	YarnUtils.java	Access, secur, privilrg, permiss	// Now remove the AM->RM token so that containers cannot access it. import java.security.PrivilegedExceptionAction; import org.apache.hadoop.yarn.security.AMRMTokenIdentifier; try { ugi.doAs(new PrivilegedExceptionAction<TimelinePutResponse>() { @Override public TimelinePutResponse run() throws Exception { return timelineClient.putEntities(entity); } }); See the License for the specific language governing permissions and * limitations under the License.		

Resource Scheduling	ApplicationMaster.java	Schedule, sched, ki, thread	private int getMinContainerMemoryIncrement(Configuration conf) { String incrementStr = conf.get("yarn.scheduler.increment-allocation-mb"); if(incrementStr == null || incrementStr.length() == 0) { incrementStr = conf.get("yarn.scheduler.minimum-allocation-mb"); } private ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());	It seems from the code that there are elements (strings) are scheduled by a scheduler to be configured string by string. So, I can say that Resource Scheduling is applied here.	Detected by our manual search and Archie
	Client.java	Ki, thread	yarnClient.killApplication(appId); while (true) { // Check app status every 1 second. try { Thread.sleep(1000); } catch (InterruptedException e) { LOG.debug("Thread sleep in monitoring loop interrupted"); }	It seems that there is a scheduler because there are a set of threads and they are checked every 1 second so there is a scheduling policy to check their status.	
Session Management	MetronRestConstants.java	Session, user	public static final String ZK_CLIENT_SESSION_TIMEOUT = "zookeeper.client.timeout.session"; public static final String SECURITY_ROLE_USER = "USER"; public static final String SECURITY_ROLE_ADMIN = "ADMIN"; public static final String USER_SETTINGS_HBASE_TABLE_SPRING_PROPERTY = "user.settings.table"; public static final String USER_SETTINGS_HBASE_CF_SPRING_PROPERTY = "user.settings.cf"; public static final String USER_JOB_LIMIT_SPRING_PROPERTY = "user.job.limit";	I think there is a Session tactic is applied in the Metron framework. The code shows that there is a client session and that session has a timeout to be terminated.	Detected by our manual search and Archie
	StellarResult.java	User, session	/** * Indicates that the user would like to terminate the session. * * @return A result indicating that the session should be terminated.	There is a session for each user and can be terminated. It might be to have Session tactic here.	
	QuitCommand.java	session	/** * A special command that allows the user to 'quit' their REPL session. * * quit	There is a session for each user and can be quit.	

			*/		
	KafkaConfig.java	session	public Map<String, Object> consumerProperties() { final Map<String, Object> props = new HashMap<>(); props.put("bootstrap.servers", environment.getProperty(MetronRestConstants.KAFKA_BROKER_URL_SPRING_PROPERTY)); props.put("group.id", "metron-rest"); props.put("enable.auto.commit", "false"); props.put("auto.commit.interval.ms", "1000"); props.put("session.timeout.ms", "30000");	There is a session for each created property and can be time out.	
	StellarInterpreterIntegrationTest.java	User, session	/** * A user should be able to define a Zookeeper URL as a property. When this property * is defined, a connection to Zookeeper is created and available in the Stellar session. */	This code does not give any indication related to the Session tactic. It just talking about Stellar session.	
Load Balancing	JSONCleaner.java	balanc	try { //cleaner.clean(jsonText); Map obj=new HashMap(); obj.put("name","foo"); obj.put("num", 100); obj.put("balance", 1000.21); obj.put("is_vip", true); obj.put("nickname",null); Map obj1 = new HashMap(); obj1.put("sourcefile", obj);	Distributing the load among several servers is shown in the previous segments of codes	Detected by our manual search and Archie
	KafkaFunctions.java	Load, server	Map<String, Object> messageAsMap; try { // transform the message to a map of fields messageAsMap = JSONUtils.INSTANCE.load(message, JSONUtils.MAP_SUPPLIER); KAFKA_PUT('topic', ["message1"], { "bootstrap.servers": "kafka-broker-1:6667" })	It seems that there is a loading process in this code to transform a message to a map of fields. I can say there is Load Balancing tactic is applied here.	

					
	TaxiiLoader.java	Load	public static void main(String... argv) throws Exception { Configuration conf = HBaseConfiguration.create(); String zkQuorum = conf.get(HConstants.ZOOKEEPER_QUORUM); String[] otherArgs = new GenericOptionsParser(conf, argv).getRemainingArgs(); CommandLine cli = TaxiiOptions.parse(new PosixParser(), otherArgs); if(TaxiiOptions.LOG4J_PROPERTIES.has(cli)) { PropertyConfigurator.configure(TaxiiOptions.LOG4J_PROPERTIES.get(cli)); } ExtractorHandler handler = ExtractorHandler.load(FileUtils.readFileToString(new File(TaxiiOptions.EXTRACTOR_CONFIG.get(cli)))); Extractor e = handler.getExtractor(); SensorEnrichmentUpdateConfig sensorEnrichmentUpdateConfig = null; if(TaxiiOptions.ENRICHMENT_CONFIG.has(cli)) { sensorEnrichmentUpdateConfig = JSONUtils.INSTANCE.load(new File(TaxiiOptions.ENRICHMENT_CONFIG.get(cli)) , SensorEnrichmentUpdateConfig.class); sensorEnrichmentUpdateConfig.updateSensorConfigs(); } Timer timer = new Timer(); if(isStixExtractor(e)) { Extractor extractor = e; TaxiiConnectionConfig connectionConfig = TaxiiConnectionConfig.load(FileUtils.readFileToString(new File(TaxiiOptions.CONNECTION_CONFIG.get(cli)))); if(TaxiiOptions.BEGIN_TIME.has(cli)) { Date d = DATE_FORMAT.parse(TaxiiOptions.BEGIN_TIME.get(cli));	It seems that there is a loading process in this code to load the file into a handler (destination).	

			connectionConfig.withBeginTime(d); }		
Time-out	Runner.java	time	ServiceInstanceBuilder<ModelEndpoint> builder = ServiceInstance.<ModelEndpoint> builder() .address(endpointUrl.getHost()) .id(containerId) .name(name) .port(endpointUrl.getPort()) .registrationTimeUTC(System.currentTimeMillis()) .serviceType(ServiceType.STATIC) .payload(endpoint) ;	It seems in the codes that Time-out tactic is applied in Metron. This is because there is a time interval for a process with a start time and end time and time unit.	Detected by our manual search and Archie
	HbaseProfilerClient.java	time	param start The start time in epoch milliseconds. * @param end The end time in epoch milliseconds. public <T> List<T> fetch(Class<T> clazz, String profile, String entity, List<Object> groups, long durationAgo, TimeUnit unit, Optional<T> defaultValue) { long end = System.currentTimeMillis(); long start = end - unit.toMillis(durationAgo); return fetch(clazz, profile, entity, groups, start, end, defaultValue); }		
	Window.java	time	predicates are true as applied to the window interval start time, then the interval is excluded. * Exclusions trump inclusions		
	WindowProcessor.java	time	public void exitTimeUnit(org.apache.metron.profiler.client.window.generated.WindowP arser.TimeUnitContext ctx) { checkForException(ctx); switch(normalizeTimeUnit(ctx.getText())) { case "DAY": stack.push(new Token<>(TimeUnit.DAYS, TimeUnit.class)); break; case "HOUR": stack.push(new Token<>(TimeUnit.HOURS, TimeUnit.class)); break; case "MINUTE":		

			stack.push(new Token<>(TimeUnit.MINUTES, TimeUnit.class)); break; case "SECOND": stack.push(new Token<>(TimeUnit.SECONDS, TimeUnit.class)); break; default: throw new IllegalStateException("Unsupported time unit: " + ctx.getText() + ". Supported units are limited to: day, hour, minute, second " + "with any pluralization or capitalization."); } } public void exitTimeAmount(org.apache.metron.profiler.client.window.generated.Windo wParser.TimeAmountContext ctx) { if(checkForException(ctx)) { return; } if(ctx.getText().length() == 0) { throwable = new IllegalStateException("Unable to process empty string."); return; } long duration = Long.parseLong(ctx.getText()); stack.push(new Token<>(duration, Long.class)); } Token<?> timeUnit = stack.pop(); Token<?> timeDuration = stack.pop(); long duration = ConversionUtils.convert(timeDuration.getValue(), Long.class); TimeUnit unit = (TimeUnit) timeUnit.getValue(); stack.push(new Token<>(unit.toMillis(duration), Long.class));		
	WindowListener.java	time	void enterTimeInterval(WindowParser.TimeIntervalContext ctx); /** * Exit a parse tree produced by the {@code TimeInterval} * labeled alternative in {@link WindowParser#time_interval}. * @param ctx the parse tree		

			*/ void exitTimeInterval(WindowParser.TimeIntervalContext ctx); /** * Enter a parse tree produced by the {@code TimeAmount} * labeled alternative in {@link WindowParser#time_amount}. * @param ctx the parse tree */ void enterTimeAmount(WindowParser.TimeAmountContext ctx); /** * Exit a parse tree produced by the {@code TimeAmount} * labeled alternative in {@link WindowParser#time_amount}. * @param ctx the parse tree */ void exitTimeAmount(WindowParser.TimeAmountContext ctx); /** * Enter a parse tree produced by the {@code TimeUnit} * labeled alternative in {@link WindowParser#time_unit}. * @param ctx the parse tree */ void enterTimeUnit(WindowParser.TimeUnitContext ctx); /** * Exit a parse tree produced by the {@code TimeUnit} * labeled alternative in {@link WindowParser#time_unit}. * @param ctx the parse tree */ void exitTimeUnit(WindowParser.TimeUnitContext ctx);		
	WindowParser.java	time	public static class TimeUnitContext extends Time_unitContext { public TerminalNode TIME_UNIT() { return getToken(WindowParser.TIME_UNIT, 0); } public TimeUnitContext(Time_unitContext ctx) { copyFrom(ctx); }		
Restart	ObjectGet.java	restart	@Stellar(namespace="OBJECT" ,name="GET" ,description="Retrieve and deserialize a serialized object from HDFS. " + "The cache can be specified via two properties in the global config: " + "\"" + ObjectGet.OBJECT_CACHE_SIZE_KEY + "\" (default " + ObjectGet.OBJECT_CACHE_SIZE_DEFAULT + ")," + "\"" + ObjectGet.OBJECT_CACHE_EXPIRATION_KEY+ "\" (default 1440). Note, if these are changed in global config, " +	It seems that Restart tactic is applied in Metron. This is because the codes show that there is a restart process should be performed when something wrong happens.	Detected by our manual search and Archie

			"topology restart is required." , params = { "path - The path in HDFS to the serialized object" } , returns="The deserialized object."		
	KafkaComponent.java	restart	public void reset() { // Unfortunately, there's no clean way to (quickly) purge or delete a topic. // At least without killing and restarting broker anyway. stop(); start(); }		
	BatchTimeoutHelper.java	restart	These lookups are fairly expensive, and changing the result values currently require * a restart of the Storm topology anyway, so this implementation caches its results for * re-reading. If you want different results, you'll need a new instance or a restart. */		
Observer/Publish-Subscribe pattern	ApplilcationMaster.java	publish	if(timelineClient != null) { YarnUtils.INSTANCE.publishApplicationAttemptEvent(timelineClient, appAttemptID.toString(), ContainerEvents.APP_ATTEMPT_START, domainId, appSubmitterUgi); } // Tracking url to which app master publishes info for clients to monitor private String appMasterTrackingUrl = "";	Observer/Publish-Subscribe pattern seems is applied in Metron because the codes show that there is a master which publish information for clients to monitor.	Detected by our manual search and Archie
	YarnUtils.java	publish	public void publishContainerStartEvent(final TimelineClient timelineClient, Container container, String domainId, UserGroupInformation ugi) { final TimelineEntity entity = new TimelineEntity(); entity.setEntityId("" + container.getId()); entity.setEntityType(ApplicationMaster.DSEntity.DS_CONTAINER.toString()); entity.setDomainId(domainId); entity.addPrimaryFilter("user", ugi.getShortUserName()); TimelineEvent event = new TimelineEvent(); event.setTimestamp(System.currentTimeMillis()); event.setEventType(ContainerEvents.CONTAINER_START.toString());		

			event.addEventInfo("Node", container.getNodeId().toString()); event.addEventInfo("Resources", container.getResource().toString()); entity.addEvent(event); catch (Exception e) { LOG.error("Container start event could not be published for " + container.getId().toString(), e instanceof UndeclaredThrowableException ? e.getCause() : e); }		
Shared-Repository pattern	PausableInput.java	reposit	/** * Repositions this stream to the position at the time the * <code>mark</code> method was last called on this input stream. * <p> /** * Marks the current position in this input stream. A subsequent call to * the <code>reset</code> method repositions this stream at the last marked * position so that subsequent reads re-read the same bytes. * <p>	It doesn't seem that reposit in the codes of PausableInput class is related to the Repository pattern. It just refere to the reset process to reposit an input stream at the last marked position. But, it might be to have Repository pattern by using Buffer to store the input streams.	Detected by our manual search and Archie
Pipes and Filters pattern	FileFilterUtileTest.java, Filters.java, FixedPcapFilterTest.java, PcapFilterTest.java, FileFilterUtil.java,	filter		It doesn't seem that pipe term in the previous code is not related to the Pipes and Filters pattern.	Detected by Archie but not by our manual search

Checkpoint	Is not detected by Archie at all	No detected keywords	-	-	Not detected by Archie and our manual search
Retry tactic	Is not detected by Archie at all	No detected keywords	-	-	Detected by our manual search but not by Archie

Tactic/Pattern	**Source file**	**Detected Terms**	**Code Where Terms are Detected**	**Description**	**Compared with My Manual Search**
Pipes and Filters	FilterFunction.java, PipelineBreakingTest.java, FilterDescriptor.java, FilterNode.java, PythoneFilterFunction.java	pipe, filter		Pipes and Filters pattern is implemented in Flink and all the previous classes are used in its implementation.	Detected by our manual search and Archie

Heartbeat	HeartbeatManagerOptions.java	Heartbeat, beat	/** * The set of configuration options relating to heartbeat manager settings. */ @PublicEvolving public class HeartbeatManagerOptions { /** Time interval for requesting heartbeat from sender side. */ public static final ConfigOption<Long> HEARTBEAT_INTERVAL = key("heartbeat.interval") .defaultValue(10000L) .withDescription("Time interval for requesting heartbeat from sender side."); /** Timeout for requesting and receiving heartbeat for both sender and receiver sides. */ public static final ConfigOption<Long> HEARTBEAT_TIMEOUT = key("heartbeat.timeout") .defaultValue(50000L) .withDescription("Timeout for requesting and receiving heartbeat for both sender and receiver sides."); // -- /** Not intended to be instantiated. */ private HeartbeatManagerOptions() {} }	HeartbeatManagerOptions class seems it managing the requesting of heartbeat from sender to receiver. This is an indication that there is a Heartbeat tactic applied in Flink.	Detected by our manual search and Archie
Resource Scheduling	StatusUpdate.java	Scheduler, resource	** * Message sent by the callback handler to the scheduler actor * when the status of a task has changed (e.g., a slave is lost and so the task is lost, * a task finishes and an executor sends a status update saying so, etc).		Detected by our manual search and Archie

			*/ public class StatusUpdate implements Serializable { private static final long serialVersionUID = 1L; private final Protos.TaskStatus status; public StatusUpdate(Protos.TaskStatus status) { this.status = status; } public Protos.TaskStatus status() { return status; }	There is a scheduler based on the comments in previous column. However, there is no a clear evidence shows that the scheduler is implemented in FLink.	
	SlaveLost.java		/** * Message sent by the callback handler to the scheduler actor * when a slave has been determined unreachable (e.g., machine failure, network partition). */ public class SlaveLost implements Serializable { private static final long serialVersionUID = 1L; private final Protos.SlaveID slaveId; public SlaveLost(Protos.SlaveID slaveId) { this.slaveId = slaveId; } public Protos.SlaveID slaveId() { return slaveId; } @Override public String toString() { return "SlaveLost{" + "slaveId=" + slaveId +		

			'}'; } }		
	ResourcOffers.java		** * Message sent by the callback handler to the scheduler actor * when resources have been offered to this framework. */ public class ResourceOffers implements Serializable { private static final long serialVersionUID = 1L; private final List<Protos.Offer> offers; public ResourceOffers(List<Protos.Offer> offers) { requireNonNull(offers); this.offers = offers; } public List<Protos.Offer> offers() { return offers; } @Override public String toString() { return "ResourceOffers{" + "offers=" + offers + '}'; } }		
	CheckpointDeclineReason.java	scheduler	/** * Various reasons why a checkpoint was declined. */ public enum CheckpointDeclineReason { COORDINATOR_SHUTDOWN("Checkpoint coordinator is shut down."),		

			PERIODIC_SCHEDULER_SHUTDOWN("Periodic checkpoint scheduler is shut down."), ALREADY_QUEUED("Another checkpoint request has already been queued."), TOO_MANY_CONCURRENT_CHECKPOINTS("The maximum number of concurrent checkpoints is exceeded"), MINIMUM_TIME_BETWEEN_CHECKPOINTS("The minimum time between checkpoints is still pending. " + "Checkpoint will be triggered after the minimum time."), NOT_ALL_REQUIRED_TASKS_RUNNING("Not all required tasks are currently running."), EXCEPTION("An Exception occurred while triggering the checkpoint."), EXPIRED("The checkpoint expired before triggering was complete"); // ----------		
	CheckpointCoordinatorDeActivator.java	scheduler	** * This actor listens to changes in the JobStatus and activates or deactivates the periodic * checkpoint scheduler. */ public class CheckpointCoordinatorDeActivator implements JobStatusListener { private final CheckpointCoordinator coordinator;		

			public CheckpointCoordinatorDeActivator(CheckpointCoordinator coordinator) { this.coordinator = checkNotNull(coordinator); } @Override public void jobStatusChanges(JobID jobId, JobStatus newJobStatus, long timestamp, Throwable error) { if (newJobStatus == JobStatus.RUNNING) { // start the checkpoint scheduler coordinator.startCheckpointScheduler(); } else { // anything else should stop the trigger for now coordinator.stopCheckpointScheduler(); } } }		
Checkpoint	Checkpoint.java	checkpoint	public class Checkpoints { private static final Logger LOG = LoggerFactory.getLogger(Checkpoints.class); /** Magic number at the beginning of every checkpoint metadata file, for sanity checks. */ public static final int HEADER_MAGIC_NUMBER = 0x4960672d; // -- // Writing out checkpoint metadata	The Checkpoint class refers to that there is a checkpoint tactic and its state is changing within an interval of time.	Detected by our manual search and Archie

			// ---------- ---------- public static <T extends Savepoint> void storeCheckpointMetadata(T checkpointMetadata, OutputStream out) throws IOException { DataOutputStream dos = new DataOutputStream(out); storeCheckpointMetadata(checkpointMetada ta, dos); }		
	CheckpointCoordinator.javas	checkpoi nt	/** * Discards the given state object asynchronously belonging to the given job, execution attempt * id and checkpoint id. * * @param jobId identifying the job to which the state object belongs * @param executionAttemptID identifying the task to which the state object belongs * @param checkpointId of the state object * @param subtaskState to discard asynchronously */ private void discardSubtaskState(final JobID jobId, final ExecutionAttemptID executionAttemptID, final long checkpointId, final TaskStateSnapshot subtaskState) { if (subtaskState != null) {		

			executor.execute(new Runnable() { @Override public void run() { try { subtaskState.discardState(); } catch (Throwable t2) { LOG.warn("Could not properly discard state object of checkpoint {} " + "belonging to task {} of job {}.", checkpointId, executionAttemptID, jobId, t2); } } }); } } }		
	CheckpointingOptions.java		/** * A collection of all configuration options that relate to checkpoints * and savepoints. */		
	Recoverab leWriter.java		* <p>These writers are useful in the context of checkpointing. The example below illustrates * how to use them: * * <pre>{@code * // --------- initial run -------- * RecoverableWriter writer = fileSystem.createRecoverableWriter(); * RecoverableFsDataOutputStream out = writer.open(path); * out.write(...);		

			* * // persist intermediate state * ResumeRecoverable intermediateState = out.persist(); * storeInCheckpoint(intermediateState); * * // --------- recovery -------- * ResumeRecoverable lastCheckpointState = ...; // get state from checkpoint * RecoverableWriter writer = fileSystem.createRecoverableWriter(); * RecoverableFsDataOutputStream out = writer.recover(lastCheckpointState); * * out.write(...); // append more data * * out.closeForCommit().commit(); // close stream and publish all the data *		
	CheckpointCoordinatorDeActivator.java	checkpoint	** * This actor listens to changes in the JobStatus and activates or deactivates the periodic * checkpoint scheduler. */ public class CheckpointCoordinatorDeActivator implements JobStatusListener { private final CheckpointCoordinator coordinator; public CheckpointCoordinatorDeActivator(CheckpointCoordinator coordinator) { this.coordinator = checkNotNull(coordinator); } @Override		

			public void jobStatusChanges(JobID jobId, JobStatus newJobStatus, long timestamp, Throwable error) { if (newJobStatus == JobStatus.RUNNING) { // start the checkpoint scheduler coordinator.startCheckpointScheduler(); } else { // anything else should stop the trigger for now coordinator.stopCheckpointScheduler(); } } }		
	CheckpointCoordinatorGatway.java	checkpoint	public interface CheckpointCoordinatorGateway extends RpcGateway { void acknowledgeCheckpoint(final JobID jobID, final ExecutionAttemptID executionAttemptID, final long checkpointId, final CheckpointMetrics checkpointMetrics, final TaskStateSnapshot subtaskState); void declineCheckpoint(JobID jobID, ExecutionAttemptID executionAttemptID, long checkpointId, Throwable cause); }		
	CheckpointDeclineReason.java	checkpoint	/** * Various reasons why a checkpoint was declined.		

			*/ public enum CheckpointDeclineReason { COORDINATOR_SHUTDOWN("Checkpoint coordinator is shut down."), PERIODIC_SCHEDULER_SHUTDOWN("Periodic checkpoint scheduler is shut down."), ALREADY_QUEUED("Another checkpoint request has already been queued."), TOO_MANY_CONCURRENT_CHECKPOINTS("The maximum number of concurrent checkpoints is exceeded"), MINIMUM_TIME_BETWEEN_CHECKPOINTS("The minimum time between checkpoints is still pending. " + "Checkpoint will be triggered after the minimum time."), NOT_ALL_REQUIRED_TASKS_RUNNING("Not all required tasks are currently running."), EXCEPTION("An Exception occurred while triggering the checkpoint."), EXPIRED("The checkpoint expired before triggering was complete"); // ----------		
Exception Handling	CliArgsException.java/ HttpRequestHandler.java/ PipelineErrorHandler.java	Exception, handle, try, catch	package org.apache.flink.client.cli; /** * Special exception that is thrown when the command line parsing fails. */		Detected by our manual search and Archie

			public class CliArgsException extends Exception { private static final long serialVersionUID = 1L; public CliArgsException(String message) { super(message); } public CliArgsException(String message, Throwable cause) { super(message, cause); } }		
Kerberos	Utils.java	Kerbero, credenti, authent	/** * Obtain Kerberos security token for HBase. */ private static void obtainTokenForHBase(Credentials credentials, Configuration conf) throws IOException { if (UserGroupInformation.isSecurityEnabled()) { LOG.info("Attempting to obtain Kerberos security token for HBase"); try { // ---- // Intended call: HBaseConfiguration.addHbaseResources(conf); Class .forName("org.apache.hadoop.hbase.HBaseConfiguration") .getMethod("addHbaseResources", Configuration.class) .invoke(null, conf); // ----		Detected by our manual search and Archie

			LOG.info("HBase security setting: {}", conf.get("hbase.security.authentication")); if (!"kerberos".equals(conf.get("hbase.security.authentication"))) { LOG.info("HBase has not been configured to use Kerberos."); return; }		
Observer/Publish-Subscribe pattern	QuarantineMonitor.java	Subscribe, listen	** * The quarantine monitor subscribes to the event bus of the actor system in which it was started. * It listens to {@link AssociationErrorEvent} which contain information if I got quarantined * or quarantine another remote actor system. If the actor detects that the actor system has been * quarantined or quarantined another system, then the {@link QuarantineHandler} is called. * * <p>IMPORTANT: The implementation if highly specific for Akka 2.3.7. With different version the * quarantine state might be detected differently.	There is publishing and subscribing method of messages is implemented in Metron. This gives us an indication that Observer/Publish-Subscribe pattern is implemented in Flink.	Detected by our manual search and Archie
	TimeoutListener.java		ublic interface TimeoutListener<K> { /** * Notify the listener about the timeout for an event identified by key. Additionally the method * is called with the timeout ticket which allows to identify outdated timeout events. * * @param key identifying the timed out event * @param ticket used to check whether the timeout is still valid */		

			void notifyTimeout(K key, UUID ticket); }		
Shared-Repository pattern	MesosServices.java	Store, storage	/** * Creates a {@link MesosWorkerStore} which is used to persist mesos worker in high availability * mode. * * @param configuration to be used * @param executor to run asynchronous tasks * @return a mesos worker store * @throws Exception if the mesos worker store could not be created */ MesosWorkerStore createMesosWorkerStore(Configuration configuration, Executor executor) throws Exception;	It seems that MesosServices.java is not related to the Shared-Repository pattern. Store in the previous codes is only a word refers to a store for mesos worker which is used to persist mesos worker in high availability.	Not detected by our manual search and Archie
	FileSystemBlobStore.java		/** * Blob store backed by {@link FileSystem}. * * <p>This is used in addition to the local blob storage for high availability. */		
	PermanentBlobCache.java		/** * Returns the path to a local copy of the file associated with the provided job ID and blob * key. * * <p>I will first attempt to serve the BLOB from the local storage. If the BLOB is not in * there, I will try to download it from the HA store, or directly from the {@link BlobServer}.		
	BlobCacheService.java		/** * The BLOB cache provides access to BLOB services for permanent and transient BLOBs. */ public class BlobCacheService implements BlobService {		

			/** Caching store for permanent BLOBs. */ private final PermanentBlobCache permanentBlobCache; /** Store for transient BLOB files. */ private final TransientBlobCache transientBlobCache; /**		
	BlobClient.java		/** * The BLOB client can communicate with the BLOB server and either upload (PUT), download (GET), * or delete (DELETE) BLOBs. */ public final class BlobClient implements Closeable {		
Active Redundancy	ExcutionGraph.java	Fault, toler, activ, topology		It seems that Active Redundancy tactic is not implemented in Flink. Archie detects it and returns java classes which have its related terms. However, no one of these classes is implementing Active Redundancy tactic.	Detected by Archie but not by our manual search
	PartioningProperities.java		for (int i = 0; i < newParallelism; ++i) {		

Load Balancing	RoundRobinOperatorStateRepartitioner.java	Load, balanc	// Preparation: calculate the actual index considering wrap around int parallelOpIdx = (i + startParallelOp) % newParallelism; // Now calculate the number of partitions I will assign to the parallel instance in this round ... int numberOfPartitionsToAssign = baseFraction; // ... and distribute odd partitions while I still have some, one at a time if (remainder > 0) { ++numberOfPartitionsToAssign; --remainder; } else if (remainder == 0) { // I are out of odd partitions now and begin our next redistribution round with the current // parallel operator to ensure fair load balance newStartParallelOp = parallelOpIdx; --remainder; }	It seems that Load Balancing tactic is not implemented in Flink. Archie detects it and returns java classes whcih have its related terms. However, no one of these classes is implementing Load Balancing tactic. This case is calles a False Positive (FP).	Detected by our manual search and Archie
Broker pattern	Kafka08PartitionDiscoverer.java	broker	* @param broker broker instance * @return Node representing the given broker */ private static Node brokerToNode(Broker broker) {		Detected by our manual search and Archie

			return new Node(broker.id(), broker.host(), broker.port()); } /** * Validate that at least one seed broker is valid in case of a * ClosedChannelException. * * @param seedBrokers * array containing the seed brokers e.g. ["host1:port1", * "host2:port2"] * @param exception * instance	It seems that there is a class called Broker. It works like a broker between clients and servers. This gives us an indication that Broker pattern is implemented in Flink.	
	SimpleConsumerThread.java		** * This class implements a thread with a connection to a single Kafka broker. The thread * pulls records for a set of topic partitions for which the connected broker is currently * the leader. The thread deserializes these records and emits them. * * @param <T> The type of elements that this consumer thread creates from Kafka's byte messages * and emits into the Flink DataStream.		
	Broker.java		/** * A concurrent data structure that allows the hand-over of an object between a pair of threads. */ public class Broker<V> {		
	IterationAggregatorBroker.java		/** * {@link Broker} for {@link RuntimeAggregatorRegistry}. */ public class IterationAggregatorBroker extends Broker<RuntimeAggregatorRegistry> {		

			private static final IterationAggregatorBroker INSTANCE = new IterationAggregatorBroker(); /** * Retrieve singleton instance. */ public static IterationAggregatorBroker instance() { return INSTANCE; } }		
	SolutionSetUpdateBarrierBroker.java		/** * Broker to hand over {@link SolutionSetUpdateBarrier} from * {@link IterationHeadTask} to * {@link IterationTailTask}. */ public class SolutionSetUpdateBarrierBroker extends Broker<SolutionSetUpdateBarrier> { private static final SolutionSetUpdateBarrierBroker INSTANCE = new SolutionSetUpdateBarrierBroker(); private SolutionSetUpdateBarrierBroker() { }		
Resource Pooling	FlinkKafkaProducer011.java	pool	* <p>In this mode {@link FlinkKafkaProducer011} sets up a pool of {@link FlinkKafkaProducer}. Between each * checkpoint there is created new Kafka transaction, which is being committed on * {@link FlinkKafkaProducer011#notifyCheckpointComplete(long)}. If checkpoint complete notifications are * running late, {@link FlinkKafkaProducer011} can run out of {@link FlinkKafkaProducer}s in the pool. In that * case any subsequent {@link FlinkKafkaProducer011#snapshotState(FunctionSnapshotContext)} requests will fail	It seems that Resource pool is used in Flink to store the producers.	Detected by our manual search and Archie

			* and {@link FlinkKafkaProducer011} will keep using the {@link FlinkKafkaProducer} from previous checkpoint. * To decrease chances of failing checkpoints there are three options: * - decrease number of max concurrent checkpoints * - make checkpoints more reliable (so that they complete faster) * - increase delay between checkpoints * - increase size of {@link FlinkKafkaProducer}s pool		
	FileCache.java	It just has an evidence about the existence of a thread pool.	public void shutdown() { synchronized (lock) { // first shutdown the thread pool ScheduledExecutorServices = this.executorService; if (es != null) { es.shutdown(); try { es.awaitTermination(cleanupInterval, TimeUnit.MILLISECONDS); } catch (InterruptedException e) { // may happen } }		
	BufferPool.java		** * A dynamically sized buffer pool. */ public interface BufferPool extends BufferProvider, BufferRecycler { /**		

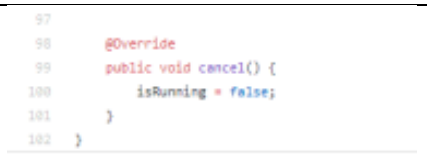
			* Destroys this buffer pool. * * <p>If not all buffers are available, they are recycled lazily as soon as they are recycled. */ void lazyDestroy(); /** * Checks whether this buffer pool has been destroyed. */ @Override boolean isDestroyed();		
	LocalBufferPool.java		class LocalBufferPool implements BufferPool { private static final Logger LOG = LoggerFactory.getLogger(LocalBufferPool.class); /** Global network buffer pool to get buffers from. */ private final NetworkBufferPool networkBufferPool;		
Authenticate	Utils.java	Credent, authent	/** * Obtain Kerberos security token for HBase. */ private static void obtainTokenForHBase(Credentials credentials, Configuration conf) throws IOException { if (UserGroupInformation.isSecurityEnabled()) { LOG.info("Attempting to obtain Kerberos security token for HBase"); try { // ---- // Intended call: HBaseConfiguration.addHbaseResources(conf); Class .forName("org.apache.hadoop.hbase.HBaseConfiguration")	It seems that Authenticate tactic is used in Flink from the previous codes.	Detected by our manual search and Archie

			.getMethod("addHbaseResources", Configuration.class) .invoke(null, conf); // ---- LOG.info("HBase security setting: {}", conf.get("hbase.security.authentication")); if (!"kerberos".equals(conf.get("hbase.security.authentication"))) { LOG.info("HBase has not been configured to use Kerberos."); return; }		
Restart	RestartStrategy.java	restart	** * Strategy for {@link ExecutionGraph} restarts. */ public interface RestartStrategy { /** * True if the restart strategy can be applied to restart the {@link ExecutionGraph}. * * @return true if restart is possible, otherwise false */ boolean canRestart();	It seems that there is restart strategy is applied in FLink in case of something wong happens. This is shown in the previous code.	Detected by our manual search and Archie

					
	RestartAllStrategy.java		import static org.apache.flink.util.Preconditions.checkNotNull; /** * Simple failover strategy that triggers a restart of all tasks in the * execution graph, via {@link ExecutionGraph#failGlobal(Throwable)}. */		
Time Stamp	Clock.java	clock, time	package org.apache.flink.streaming.connectors.fs; /** * A clock that can provide the current time. * * * <p>Normally this would be system time, but for testing a custom {@code Clock} can be provided. */ public interface Clock { /** * Return the current system time in milliseconds. */ long currentTimeMillis(); }	It seems that there is a clock interface that control the execution of the processes and provide the system time. This gives us an indication that Time Stamp is implemented in Flink.	Time Stamp Detected by our manual search and Archie
Layers	JobGraphTest.java	layer	public void testTopologicalSort2() { try {		

			JobVertex source1 = new JobVertex("source1"); JobVertex source2 = new JobVertex("source2"); JobVertex root = new JobVertex("root"); JobVertex l11 = new JobVertex("layer 1 - 1"); JobVertex l12 = new JobVertex("layer 1 - 2"); JobVertex l13 = new JobVertex("layer 1 - 3"); JobVertex l2 = new JobVertex("layer 2");	From the previous codes and the general architecture of Flink, it seems that FLink is using Layers pattern.	Detected by our manual search and Archie
	PythonApplyFunction.java		/** * The {@code PythonApplyFunction} is a thin wrapper layer over a Python UDF {@code WindowFunction}. * It receives an {@code WindowFunction} as an input and keeps it internally in a serialized form. * It is then delivered, as part of job graph, up to the TaskManager, then it is opened and becomes * a sort of mediator to the Python UDF {@code WindowFunction}. * * This function is used internally by the Python thin wrapper layer over the streaming data * functionality */ public class PythonApplyFunction<W extends Window> extends AbstractPythonUDF<WindowFunction<PyObject, Object, Object, W>> implements WindowFunction<PyObject, PyObject, PyKey, W> { private static final long serialVersionUID = 577032239468987781L; private transient PythonCollector collector;		

			public PythonApplyFunction(WindowFunction<PyObject, Object, Object, W> fun) throws IOException { super(fun); }		
Ping/Echo	NetworkFailuresProxyTest.java	echo	EchoServer echoServer = new EchoServer(SOCKET_TIMEOUT); public EchoClient(String hostName, int portNumber, int socketTimeout) throws IOException { socket = new Socket(hostName, portNumber); socket.setSoTimeout(socketTimeout); output = new PrintWriter(socket.getOutputStream(), true); input = new BufferedReader(new InputStreamReader(socket.getInputStream())); }	Echo in the previous codes is only a word. It doesn't refers to Ping/Echo tactic.	Detected by our manual search and Archie
Session Management	YarnSessionCapacitySchedulerITCase.java/YarnSessionFIFOIT Case.java/YarnTestBase.java	session	* <p>This class is located in a different package which is build after flink-dist. This way, * I can use the YARN uberjar of flink to start a Flink YARN session. * * <p>The test is not thread-safe. Parallel execution of tests is not possible! */ public abstract class YarnTestBase extends TestLogger {	There is a session for each task .	Detected by our manual search and Archie
Time Out	JobGraph.java	timeout	/** * Sets the timeout of the session in seconds. The timeout specifies how long a job will be kept * in the job manager after it finishes. * @param sessionTimeout The timeout in seconds */ public void setSessionTimeout(long sessionTimeout) {	It seems that there is a Time-out tactic since the previous code is talking about time out, which specifies how long a job will be kept in the job manager after it finishes.	Detected by our manual search and Archie

			this.sessionTimeout = sessionTimeout; } }		
PBAC	NetUtils.java	oc		It is only oc word. It is not related to PBAC tactic any more.	Detected by Archie but not by our manual search
RBAC	NetUtils.java	access		It is only access word. It is not related to RBAC tactic any more.	Detected by Archie but not by our manual search
Audit Trail	Configuration.java	log		Log word is not related to Audit Trail in the previous code.	Detected by Archie but not by our manual search
Cancel	WikipediaEditsSource.java	cancel	 @Override public void cancel() { isRunning = false; } }	Cancel tactic is implemented in Flink through Cancel() method as shown in the pervious code.	Detected by our manual search and Archie