symmetry

MDPI

Article

Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures

Miloš Milić [1,*] and Dragana Makajić-Nikolić [2]

[1] Department of Software Engineering, Faculty of Organizational Sciences, University of Belgrade, Jove Ilića 154, 11000 Belgrade, Serbia
[2] Department of Operation Research and Statistics, Faculty of Organizational Sciences, University of Belgrade, Jove Ilića 154, 11000 Belgrade, Serbia
* Correspondence: mmilic@fon.bg.ac.rs; Tel.: +381-113950877

Abstract: Various architectures can be applied in software design. The aim of this research is to examine a typical implementation of Jakarta EE monolithic and microservice software architectures in the context of software quality attributes. Software quality standards are used to define quality models, as well as quality characteristics and sub-characteristics, i.e., software quality attributes. This paper evaluates monolithic and microservice architectures in the context of Coupling, Testability, Security, Complexity, Deployability, and Availability quality attributes. The performed examinations yielded a quality-based mixed integer goal programming mathematical model for software architecture optimization. The model incorporates various software metrics and considers their maximal, minimal or targeted values, as well as upper and lower deviations. The objective is the sum of all deviations, which should be minimal. Considering the presented model, a solution which incorporated multiple monoliths and microservices was defined. This way, the internal structure of the software is defined in a consistent and symmetrical context, while the external software behavior remains unchanged. In addition, an intersection point of monolithic and microservice software architectures, where software metrics obtain the same values, was introduced. Within the intersection point, either one of the architectures can be applied. With the exception of some metrics, an increase in the number of features leads to a value increase of software metrics in microservice software architecture, whilst these values are constant in monolithic software architecture. An increase in the number of features indicated a quality attribute's importance for the software system should be examined and an appropriate architecture should be selected accordingly. Finally, practical recommendations regarding software architectures in terms of software quality were given. Since each software system needs to meet non-functional in addition to functional requirements, a quality-driven software engineering can be established.

Keywords: software architecture; monolithic architecture; microservice architecture; Jakarta EE; software quality; quality attribute; quality-based model; architecture optimization; intersection point

Citation: Milić, M.; Makajić-Nikolić, D. Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures. *Symmetry* **2022**, *14*, 1824. https://doi.org/10.3390/sym14091824

Academic Editor: Roman Tsarev

Received: 1 August 2022
Accepted: 22 August 2022
Published: 2 September 2022

1. Introduction

Software engineering is a discipline that considers all aspects of the software development process, which can apply various models, methods, strategies, and activities [1–3]. In that sense, one of the most important activities of the software development process is software design. Software design can be defined as an activity in which software requirements are analyzed in order to produce a description of the software's internal structure [4]. As a result, software architecture is created and used to describe the system's design at a high level [5]. Thus, software architecture represents the basis for further software development processes.

Selection of software architecture guides the software development process. Software is constructed based on the software architecture [4], which defines the software system's components, as well as the interfaces between them [6]. Each software system's component should be tested (i.e., unit and integration testing of components) [1]. Additionally, a software system's components should further be maintained and reused. Thus, it is of utmost importance to determine a software architecture that best suits the software system in question.

Numerous software architectures could be applied in the software development process [7,8]. Software architectures are subject to quality evaluation. As software systems should assure some quality characteristics, the architecture chosen for the system has to be compliant with the defined quality model. The aim of this research is to consider monolithic and microservice software architectures in the context of software quality attributes. Therefore, the evaluation process examined a software system for project assessment, while the software development process applied Jakarta EE (Jakarta Enterprise Edition) web technology stack and Spring Framework. Since measured values could differ, research results can contribute to a proper choice of architecture that best suits the specific software system.

In software design, microservice architecture is one of the alternatives to a monolithic architecture. Software quality standards are used to define quality models, while quality models define quality attributes. The evaluation process of monolithic and microservice architectures considered Coupling, Testability, Security, Complexity, Deployability, and Availability quality attributes. The performed evaluation produced a quality-based mathematical model for software architecture optimization. The model considers features of a software system and the importance of the selected software metrics. In this context, the structure of the solution, which incorporates multiple monoliths and multiple microservices, was defined, i.e., the proposed model considers the structure of a software system in which monolithic and microservice architectures coexist. In addition, an intersection point of monolithic and microservice software architectures, where software metrics obtain the same values, was introduced. Finally, the research presents practical recommendations regarding the examined architectures in the context of quality attributes. Taking into account that the proposed model considers various aspects of software quality in terms of quality attributes and metrics, the internal structure of the software can be defined in a consistent manner.

This paper is organized as follows. Section 2 outlines monolithic and microservice architectures, as well as software quality standards. The section also gives an overview of some quality attributes which need to be considered in the software development process. Section 3 examines monolithic and microservice architectures in the context of software quality attributes. The quality-based model for software architecture optimization is presented in Section 4. Threats to validity are presented in Section 5. Discussion is presented in Section 6. Section 7 summarizes the research results, main conclusions, and presents limitations and further research directions.

2. Background

This section outlines monolithic and microservice software architectures and introduces software quality standards and software quality attributes.

2.1. Monolithic and Microservice Software Architectures

In general, architecture could be defined as the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment [9]. Architecture also includes the principles for component design and evolution. On the other hand, software architecture could be defined as a set of structures needed to reason about a system, which comprises software elements, relations among them, and properties of both [10]. Software architecture is defined during the software design process [4].

A monolith is a software application whose modules cannot be executed independently [11]. Figure 1 depicts a conceptual overview (one possible solution) of monolithic application. A monolithic application consists of multiple modules where each module has its own responsibility [12]. Module 1 may contain user interface, Module 2 may contain components that refer to application's business logic, Module 3 may contain data access components, etc. Figure 1 also shows that a monolithic application usually has only one database.

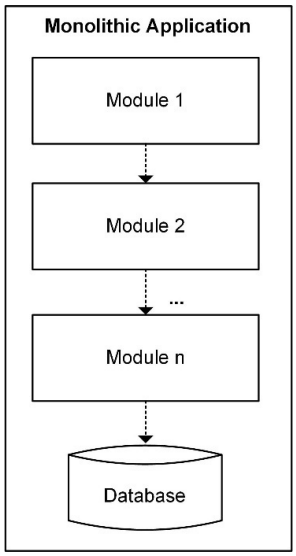

Figure 1. Conceptual overview of monolithic application.

Dependencies between modules imply that a monolithic application is compiled and assembled as an entity. Therefore, a change in one module indicates that the module should be recompiled and that the whole should be reassembled [13].

Microservice architecture is one of the alternatives to monolithic architecture. When large monolithic software systems are observed, adding new functionalities may affect the complexity and maintenance of software. The solution can be to implement microservice architecture, which is aimed at grouping functionalities in a set of small services that cooperate with each other. Microservice architecture is a distributed application where all its modules are microservices [11]. A microservice is a cohesive, independent process which interacts via messages [11]. Figure 2 provides a conceptual overview of microservice application. The figure illustrates multiple independent microservices, while each microservice has its own responsibility [14]. All presented elements are parts of a microservice application. Moreover, each microservice (i.e., each module) may contain a user interface, a component that refers to an application's business logic, a component for data access, as well as other components. Figure 2 also shows that each microservice has its own database.

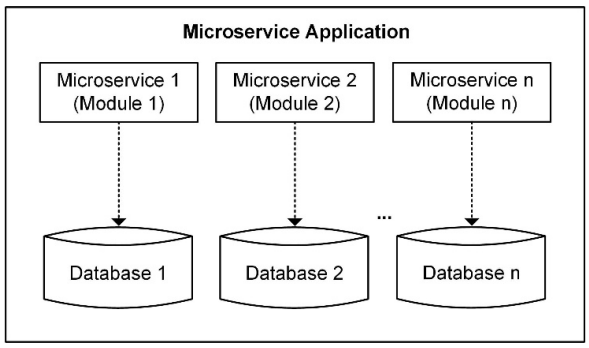

Figure 2. Conceptual overview of microservice application.

Independence among microservices indicates that each microservice is separately compiled and assembled. Therefore, a change in one microservice does not require recompiling and reassembling of the whole application [13]. In addition, each software development team must have only the source code of a microservice for which it is directly responsible (i.e., the microservice delegated to that team).

2.2. Software Quality Standards

Taking into account the complexity of modern software systems and their application in various fields, software quality is identified as one of the knowledge areas within the Software Engineering Body of Knowledge [4]. Therefore, quality management and quality control should be considered in all phases of the software development process [15].

Quality could be defined as a product's compliance with the product's detailed specification [16]. Software quality can be defined as the capability of a software product to meet the stated and implied needs in particular circumstances [4,17], i.e., as an effective software process which results in a useful software product that provides measurable value for those who produce it and those who use it [1].

The abovementioned definitions indicate that software quality could be observed from different perspectives. Software engineers are focused on developing a software that is functional, testable, reliable, and maintainable, while a system's end users are more concerned with characteristics such as productivity, usability, effectiveness, etc. These are just some of the software quality attributes which need to be taken into account during software development.

Software quality standards are applied in the software quality evaluation process. Various quality standards are available: some represent general-purpose standards and can be used to evaluate any software system (e.g., ISO/IEC 9126 [18] and ISO/IEC 25000 [17]), while others are specialized standards applied within specific industries.

Software quality standards are used to determine the specification of a software quality model, which is the basis for software quality evaluation. However, a quality model provides no specification of how to measure value—it only determines quality characteristics and decomposes them to the level of sub-characteristics. Quality characteristics and sub-characteristics are used to identify key attributes of software quality [1], which are then applied for quality requirements specification [19]. Figure 3 illustrates the relationship between software quality standards, software quality models, and software quality attributes.

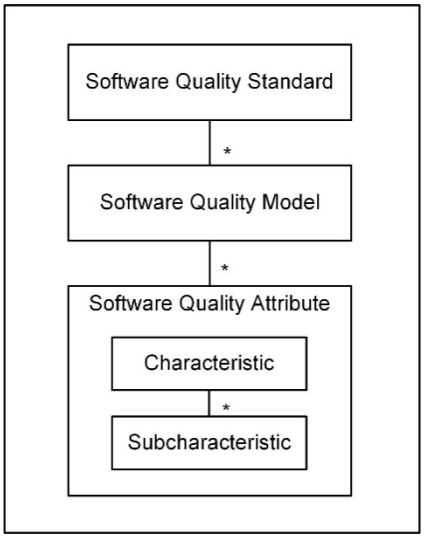

Figure 3. Relationship between software quality standards, software quality models, and software quality attributes.

It is implied above that a quality model may define multiple attributes, characteristics and sub-characteristics typically organized as a hierarchy [20]. In that context, software metrics are applied for operational measurements of the software quality characteristics and sub-characteristics. Each quality model can incorporate various metrics for quality evaluation [1,21]. A software metric can be defined as a quantitative measure of the degree to which a system, component or process possesses the observed attribute [9,20]. It is a characteristic of the software system, system's documentation or system's development process that can be objectively measured [15], based on which various aspects of software quality could be determined [22].

Different quality standards can define different quality attributes [17,18]. In this research, different terms are found for the same software quality attribute [23]. In this context, the following quality attributes should be considered in the software quality evaluation process:

- Coupling—A software system may encompass multiple components grouped in modules. Components of one module can invoke the components from another module. Coupling can be defined as a measure of interdependence among modules in a software system [4]. In that context, the coupling between components and modules should be reduced to a minimum.
- Testability—Apart from the programming code, a software system can contain numerous tests (e.g., unit tests, components tests, integration tests). Testability could be defined as a degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component and tests can be performed to determine whether those criteria have been met [17]. High level of testability is a required characteristic within the software development process.
- Security—Security considers the ways in which a software system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization [17]. Taking into consideration the importance of information and data found in a software system, security should be maintained at a high level.
- Complexity—As previously stated, a software system incorporates multiple components. Complexity could be defined as the degree to which a system or component has a design or implementation that is difficult to understand and verify [9]. A low level of complexity is a required characteristic within the software development process.
- Deployability—Subsequent to the implementation or change in some feature, a software system should be deployed to the appropriate environment (e.g., test environment, staging environment, production environment). Deployability considers all artefacts and activities required to put a software system into operation [9]. A high level of software system deployability is required.
- Availability—Availability could be defined as the degree to which a system or component is operational and accessible when required for use [17]. A software system should preserve a high level of availability.

These are only some of the quality attributes that should be examined during the software development process. This enables the fulfillment of non-functional requirements, and thus it can be stated that software quality attributes correspond to non-functional requirements of the software system [24].

3. Evaluation of Monolithic and Microservice Software Architectures

Monolithic and microservice software architectures can be evaluated in the context of different software quality attributes [13,14,23,25]. This research examines Coupling, Testability, Security, Complexity, Deployability, and Availability quality attributes (see Figure 4). These quality attributes were supported by software quality analysis tools used in the evaluation process.

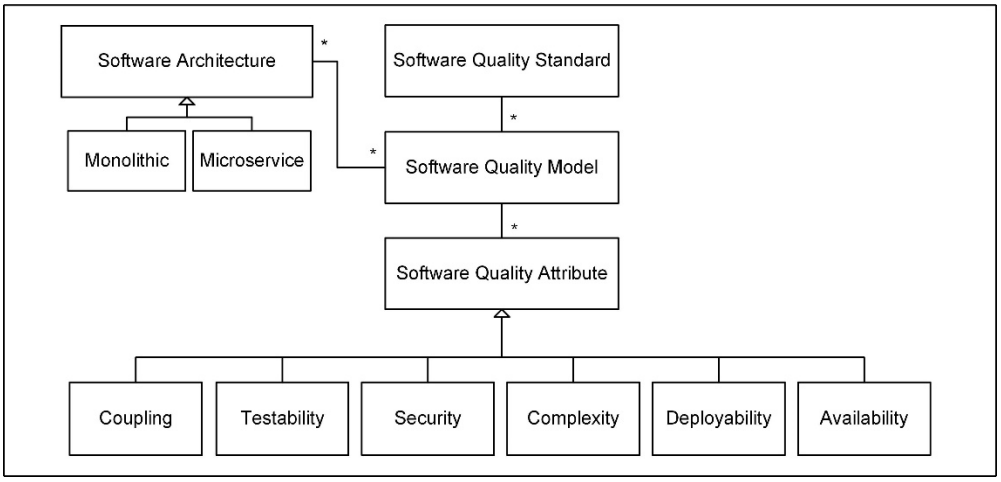

Figure 4. Relationship between software architectures and software quality attributes.

3.1. Technology Stack

Jakarta EE web technology stack was applied in the software development process. More specifically, Spring Framework 5 and Spring-related libraries (via Spring Boot 2) were applied in the back-end development, while the front-end development was performed by applying the Vue.js framework. Apache Maven 3.6 was used as a build management tool. Java web applications are usually deployed as *jar* or *war* files and are executed on the appropriate server. These files contain compiled code, external libraries and other resources needed for an application's execution. Additionally, this type of distribution enables continuous delivery and deployment and promotes DevOps software development [26]. BitBucket Pipelines were applied in DevOps software development and applications were executed on AWS (Amazon Web Services) instances.

3.2. Data Analysis

SonarQube tool (i.e., SonarCloud version) was applied in the software quality evaluation process, which enables an overview of software quality from different perspectives. During the evaluation process, *The Sonar Way* quality model was applied. Jacoco Code Coverage tool, easily integrated within the SonarQube tool, was applied in code coverage analysis. Software metrics related to Deployability were obtained from BitBucket Pipelines, while the Availability metrics were obtained from the Spring Actuator library.

Domain classes and Data Transfer Object (DTO) classes were excluded from the software quality evaluation process. These classes do not contain any of the business logics. However, they are used for data transfer and represent value classes (i.e., they just contain public constructors and getter/setter methods).

3.3. Evaluation

Feature Driven Development (FDD) was applied in the software development process. FDD is a software development method aimed at making progress on features [27]. Feature can be defined as a functional characteristic of a system of interest that end-users and other stakeholders can understand [28]. This way, functionalities are combined into a single business capability [11]. The software sub-system for project assessment was examined (shown in Figure 5) and the following features were identified: Client Management Feature (i.e., Feature A), Project Management Feature (i.e., Feature B), Employee Management Feature (i.e., Feature C), Location Management Feature (i.e., Feature D), and Task Management Feature (i.e., Feature E). Each feature contains components related to entity manipulation, i.e., create, retrieve, update and delete operations (CRUD operations). From a software engineering perspective, CRUD operations are typical operations supported by all data structures and entities [4].

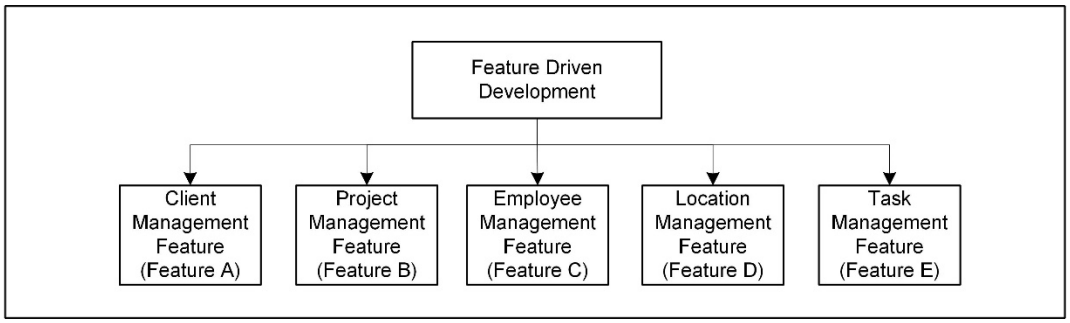

Figure 5. The examined software system features.

Monolithic software architecture presented in Figure 6 encompasses a Web Module, Service Module and Repository Module (i.e., Data Access Module). User interface invokes controller components, while the controllers invoke business services. Finally, the services invoke components within the Repository Module. The system contains one database, which is accessed from the Repository Module by applying appropriate data persistence technology (i.e., JDBC API). The application was executed on an AWS instance.

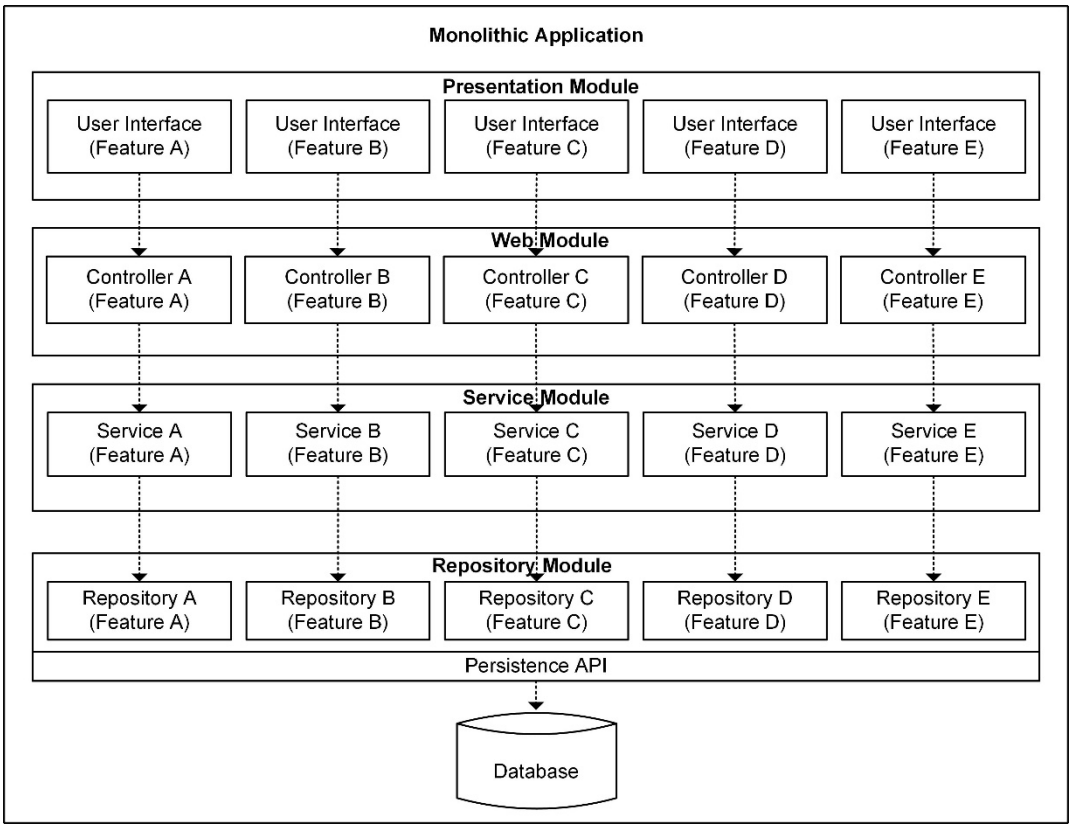

Figure 6. Monolithic software architecture.

Provided that only one Feature A exists, user interface (i.e., HTML elements and the JavaScript code) and *Controller A* will be created within Web Module (user interface invokes *Controller A*). In addition, *Controller A* component invokes *Service A* from the Service Module, which further invokes *Repository A* data access component. All of these components will be compiled and assembled into one application which produces the required Feature A. Adding a new Feature B activates the Single Responsibility Principle which states that a component should have only one reason to change [12]. In other words, each component within the software system should have one responsibility. In cases when one component has multiple responsibilities, multiple reasons for changing it may exist.

Thus, a user interface for the new Feature B is created, as well as specific *Controller B*, *Service B*, and *Repository B* components on the Web Module, Service Module and Repository Module, respectively. All created components will also be compiled and assembled into one application used for realization of both Feature A and Feature B.

The above principle was applied in the development of the remaining features (i.e., Feature C, Feature D, and Feature E). Therefore, it can be concluded that one monolithic application incorporates all features. Put differently, if a new feature is added or an existing feature is modified, some modules should be recompiled and the whole application should be reassembled. Taking into account the represented monolithic architecture, Table 1 presents the software metrics' values for the observed software quality attributes.

Table 1. Software metrics' values—Monolithic software architecture.

Quality Attribute	Software Metric	Value
Coupling	Coupling Between Objects	16
Testability	Unit Tests	90
	Code Coverage (%)	94.2
	Covered Conditions	30
	Uncovered Conditions	0
Security	Security Hotspots	3
	Security Configurations	1
Complexity	Cyclomatic Complexity	75
	Cognitive Complexity	15
Deployability	Build Time (min)	02:07
	Deployment Pipelines	1
	Deployable Size (MB)	41.2
	AWS Instances	1
	Deployment Profiles	3
Availability	Healthcheck Endpoints	1

Let us examine the microservice software architecture presented in Figure 7. As previously stated, Jakarta EE and Spring Framework were used for application development. More specifically, Spring Cloud library was applied to ensure the discovery and management of microservices. Spring Cloud incorporates the following components:

- API Gateway—provides an effective way to route to APIs (i.e., microservices);
- Service Registry—used for microservice registration and management;
- Messaging—enables asynchronous microservice communication via messaging interfaces;
- Spring Cloud Sleuth—enables auto-configuration for distributed tracing.

These are typical components in the implementation of microservice architecture [29] and are provided out-of-the-box by Spring Framework and Spring Cloud. Figure 7 shows Web Layer, Service Layer and Repository Layer (i.e., Data Access Layer) as components of microservice architecture. User interface invokes controller components, while the controllers invoke business services. Finally, the services invoke components within Repository Layer. Each microservice within the system has a separate database which is accessed from Repository Layer by applying the appropriate data persistence technology (i.e., JDBC API). Each microservice is executed on a separate AWS instance.

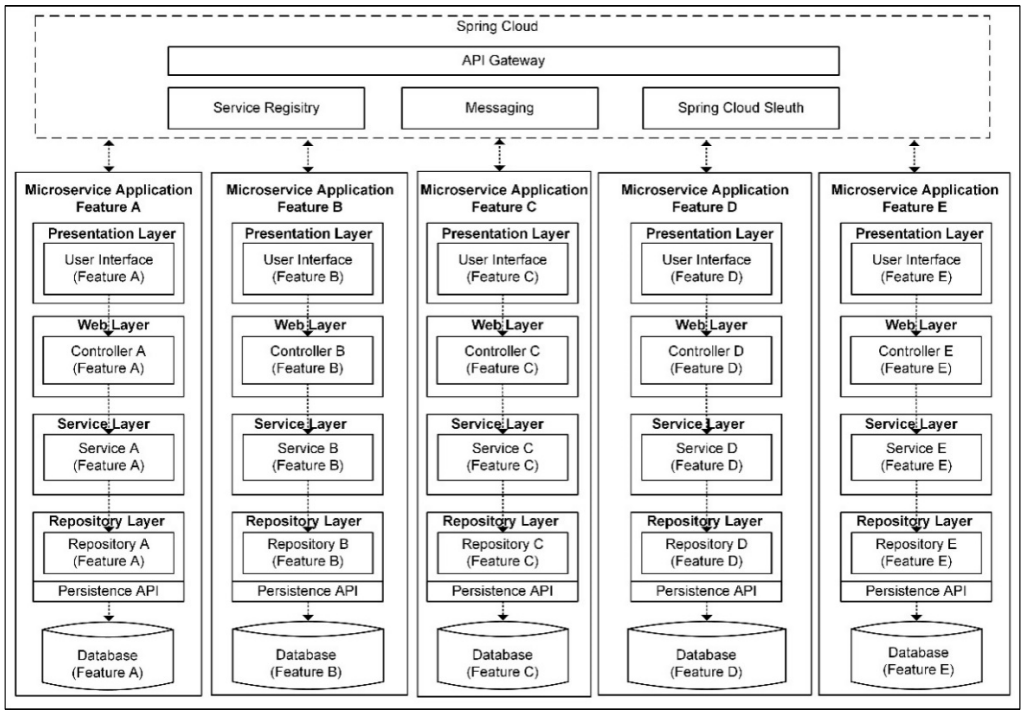

Figure 7. Microservice software architecture.

Assuming only one Feature A exists, the user interface (i.e., HTML elements and the JavaScript code) and *Controller A* will be created within Web Layer (user interface invokes *Controller A*). In addition, *Controller A* component invokes *Service A* from the Service Layer that further invokes *Repository A* data access component. These components will be compiled and assembled into one application with which the required Feature A is realized. There is a specific database for Feature A. In case of adding a new Feature B, another microservice application is designed to incorporate user interface for the new Feature B, as well as separate *Controller B*, *Service B*, and *Repository B* component within Web Layer, Service Layer, and Repository Layer, respectively. All of the created components will also be compiled and assembled into one application used for Feature B realization. A separate database for Feature B is obtained.

The abovementioned principle was used in the development of the remaining features (i.e., Feature C, Feature D, and Feature E). This implies that each feature represents a separate microservice application. In other words, if a new feature is added or an existing feature is modified, only one microservice application should be recompiled and reassembled. Table 2 presents software metrics' values for the observed software quality attributes of the microservice architecture.

The following subsections discuss software quality attributes and metrics.

3.3.1. Coupling

In terms of the Coupling attribute, the metric Coupling Between Objects was examined. An object is coupled to another one if it invokes the other one's functions [21]. User interface components invoke Controller components, while Controller components invoke Service components that further invoke Repository components. Additionally, Repository components invoke database functions. This promotes the separation of concerns between user interface, business logic, and data persistence [30].

Monolithic application contains all components, which is why the value of the metric Coupling Between Objects equals 16. However, microservice architecture contains only components required for the observed feature's realization. Thus, the value of the metric Coupling Between Objects within microservice applications equals 4. Therefore, the value of Coupling within a microservice architecture is lower than that in a monolithic architecture.

Table 2. Software metrics' values—Microservice software architecture.

Quality Attribute	Software Metric	Feature A	Feature B	Feature C	Feature D	Feature E
Coupling	Coupling Between Objects	4	4	4	4	4
Testability	Unit Tests	18	18	18	18	18
	Code Coverage (%)	87.5	86.3	87.5	86.3	84.9
	Covered Conditions	6	6	6	6	6
	Uncovered Conditions	0	0	0	0	0
Security	Security Hotspots	3	3	3	3	3
	Security Configurations	1	1	1	1	1
Complexity	Cyclomatic Complexity	18	20	18	20	21
	Cognitive Complexity	3	3	3	3	3
Deployability	Build Time (min)	01:28	01:37	01:24	01:27	01:30
	Deployment Pipelines	1	1	1	1	1
	Deployable Size (MB)	40.4	53.2	40.4	53.2	53.2
	AWS Instances	1	1	1	1	1
	Deployment Profiles	3	3	3	3	3
Availability	Healthcheck Endpoints	1	1	1	1	1

3.3.2. Testability

In terms of Testability, the code coverage above 84% was achieved for each application. Monolithic application contains 90 tests, while each microservice has 18 tests. Condition coverage of 100% was achieved. However, microservices have fewer conditions needed to be covered (each microservice application has six conditions, while a monolithic application contains 30 conditions). The results are expected since each microservice incorporates only one feature.

An interesting observation in monolithic architecture is that Repository Module, which incorporates data persistence-related components, has the code coverage of 0%. So, unit tests for each module's component were introduced and successfully passed. Further code inspection has led to the conclusion that this module contains only interfaces used for specifying data persistence methods. During the compile-time, the implementation of those interfaces was not included in this module. However, the implementation is dynamically included during the run-time through the application of aspect-oriented programming concepts. Since the module contains only interfaces, the code coverage cannot be determined. A similar conclusion could be drawn for testing the repository layer components in microservices.

The highest code coverage was achieved within a monolithic architecture (coverage of 94.2%), while the coverage for microservices was lower. Detailed code inspection has shown that the lower coverage achieved in microservices was related to microservices' communication. Microservices define only communication interfaces, while the implementation of those interfaces is provided in the run-time through the use of Spring Cloud library. An interesting conclusion is that the independent microservices (in terms of microservices' communication) had the highest code coverage: Feature A and Feature C with respective code coverage of 87.5%. On the other hand, microservices Feature B and Feature D are related to previously mentioned microservices and had coverage of 86.3%, while the most complex microservice (in terms of communication with other microservices) Feature E resulted in code coverage of 84.9%. So, as for Testability, a monolithic application contains all components and tests, which results in simpler integration testing. On the other hand, each microservice incorporates only one feature and contains only components and tests related to this particular feature, which results in more difficult integration testing.

3.3.3. Security

As for Security, we examined Common Weakness Enumeration (CWE) and Open Web Application Security Project (OWASP) recommendations that can lead to serious vulnerabilities in software. In that regard, monolithic application has three Security Hotspots. Closer code inspection revealed that these refer to safe use of command line arguments, proper configuration of Cross-Site Request Forgery, and appropriate use of user roles. Given the fact that functionalities are provided through web services and that user roles are well defined, we believe that detected hotspots pose no problems at the moment. However, these truly are potential threats which need to be taken into consideration during the software development process.

When considering microservices, we have come to some interesting conclusions. We had determined that each microservice contains three Security Hotspots and that those were the same Hotspots previously identified in the monolithic application. Additionally, according to the metric Security Configurations, each microservice has its own security mechanism whose issues need to be considered. On the other hand, secure communication between microservices should be ensured which requires additional effort in the software development process.

3.3.4. Complexity

In terms of Complexity, we examined the Cyclomatic Complexity (CC) metric. CC can be defined as the number of linearly independent paths comprised within a method [21]. This enables the determination of the software system complexity. With monolithic application, the Cyclomatic Complexity metric has a value of 75. On the other hand, Cyclomatic Complexity metric for microservice applications ranges from 18 to 21.

Cognitive Complexity metric was also examined. This software metric is the measure of understandability of the control flow statements within a software system [31]. Cognitive Complexity metric has a value of 15 in monolithic application, while this value is 3 when each microservice is observed.

As seen above, monolithic application incorporates all features, hence its Complexity is high. On the other hand, each microservice incorporates only one feature, hence its Complexity is lower than that in monolithic application.

3.3.5. Deployability

Average Build Time metric was examined in Deployability. This metric is the time (in minutes) needed for the production of deployable artefacts and includes phases of validation, code compilation, test compilation, test execution, static code analysis and packaging of the application into a deployable format. It is important to note that this process is executed within BitBucket Cloud Pipelines so the background hardware configuration remains unknown. The average build time after running pipelines multiple times was examined for each application.

When monolithic application is observed, the Average Build Time metric equals 02:07 min. On the other hand, the Average Build Time metric for a microservice application is the interval from 01:24 to 01:37 min. Therefore, the average build time of the microservice is shorter than the average build time of the monolithic application. In the event of a feature change, microservice's deployable can be deployed more quickly than that of monolithic application. Furthermore, examination of the metric Deployment Pipelines shows that each application contains one pipeline. This enables parallel execution of microservices' pipelines which additionally reduces the delivery time.

Moreover, Deployable Size metric, with value expressed in megabytes, was examined. The value of Deployable Size metric for a monolithic application is 41.2 MB, while the value of Deployable Size metric for microservice applications is the interval from 40.4 MB to 53.2 MB. This is an interesting result because monolithic application incorporates all features, while each microservice incorporates only one feature. Inspection of

pom.xml configuration files brought us to a conclusion that each deployable contains the following dependencies:

- database driver;
- spring-related libraries;
- data persistence-related libraries;
- tests-related libraries.

All the above-mentioned dependencies are incorporated into one deployable within the monolithic application. When microservices are concerned, dependencies are copied to each microservice application which results in a high value of Deployable Size metric. So, as for dependency management, it can be concluded that microservice software architecture violates the principle "Don't Repeat Yourself" (DRY) and promotes the principle "Write Everything Twice" (WET). However, due to the independency of microservices, we believe that this redundancy is required.

Finally, artefacts should be deployed to the appropriate environment. Thus, for each application one AWS instance and three profiles were defined (*application.properties* profile that can be used in the software development process, *application-test.properties* profile that can be used in a testing environment, and *application-prod.properties* profile that can be used in production). Here, each and every application will have the value of a metric AWS Instances set to 1, while the Deployment Profiles metric's value will be 3.

3.3.6. Availability

Healthcheck Endpoints metric was examined when considering Availability. These endpoints are exposed through REST web services and return a JSON-formatted response of an application's status (i.e., available, unavailable, failure). Cloud instances periodically invoke this service and, depending on the response status, relevant action can be performed. This ensures the discovery of a service [32]. In that regard, Healthcheck Endpoints metric will have a value of 1 for each and every application. This favors microservice software architecture: each microservice can independently be health-checked, based on which a new microservice instance can be started. When high-availability of a service is needed, auto-provisioning and scaling could be performed.

4. Quality-Based Model for Software Architecture Optimization

This section introduces a quality-based mathematical model for software architecture optimization. It also introduces a MicroMono application which is based on the premises of the presented model.

4.1. The Model for Software Architecture Optimization

The comparison of monolith and microservice software architectures presented in Section 3 has led to the conclusion that the software system contains five features. However, different software systems may contain a different number of features. After changing or adding a new feature, a quality analysis could be conducted as well as an examination of how the change would affect the rest of the software system. To this end, a continuous quality assessment can be performed [33] and a software metric's values can be examined. The relationship between the number of features and the change in software metrics is presented in Table 3.

A quality-based mathematical model for software architecture optimization was introduced based on the presented evaluation. The UML class diagram for the model construction is illustrated in Figure 8. Key entities, their attributes, and relationships with other entities can be detected: one software architect can define multiple software systems (i.e., problems), and the software system provides multiple features. These definitions help produce a solution. One solution can incorporate multiple monoliths and multiple microservices: a monolith contains multiple features, while a microservice contains only one feature.

Table 3. Continuous quality assessment of the software system.

	Monolithic Software Architecture					Microservice Software Architecture					
No. of Features / **Software Metric**	**1**	**2**	**3**	**…**	***k***	**1**	**2**	**3**	**…**	***k***	**Δ (Microservice − Monolithic)**
Coupling Between Objects	4	7	10	...	$3{\cdot}k + 1$	4	8	12	...	$4{\cdot}k$	$k - 1$
Unit Tests	18	36	54	...	$18{\cdot}k$	18	36	54	...	$18{\cdot}k$	0
Covered Conditions	6	12	18	...	$6{\cdot}k$	6	12	18	...	$6{\cdot}k$	0
Cognitive Complexity	3	6	9	...	$3{\cdot}k$	3	6	9	...	$3{\cdot}k$	0
Security Hotspots	3	3	3	...	3	3	6	9	...	$3{\cdot}k$	$3{\cdot}(k - 1)$
Security Configurations	1	1	1	...	1	1	2	3	...	k	$k - 1$
Deployment Pipelines	1	1	1	...	1	1	2	3	...	k	$k - 1$
AWS Instances	1	1	1	...	1	1	2	3	...	k	$k - 1$
Deployment Profiles	3	3	3	...	3	3	6	9	...	$3{\cdot}k$	$3{\cdot}(k - 1)$
Healthcheck Endpoints	1	1	1	...	1	1	2	3	...	k	$k - 1$

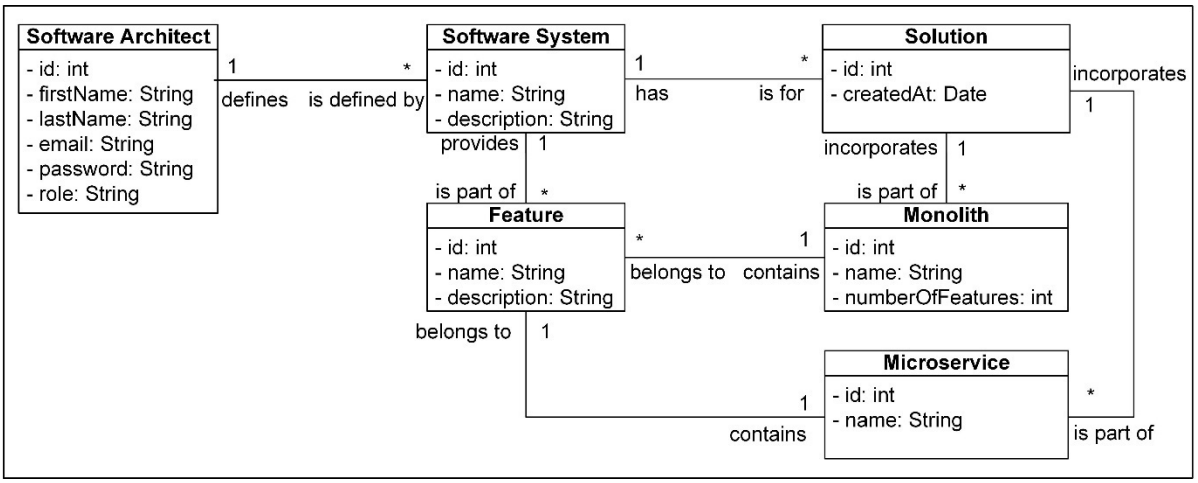

Figure 8. The class diagram for the quality-based model construction.

The observed problem is formulated as a clustering problem with additional constraints. The assumptions of the defined problem are:

- The feature that is the center of a cluster and all the features assigned to that cluster form one monolith. In this application, the center of the cluster is not relevant, but what is important is which features are grouped into a cluster (monolith);
- A cluster with one feature is a microservice;
- For each metric, maximal, minimal or targeted value is defined;
- Deviations from the maximal, minimal and target values of the metrics are allowed but should be minimized.

The software metric Coupling Between Objects considers relationships between the components of a software system. Our model applies the Model-View-Controller (MVC) paradigm, which promotes the separation of concerns between user interface, business logic and data persistence [4,30]. In this context, coupling implies the following relationships between the components: (a) User Interface–Controller, (b) Controller–Service, (c) Service–Data Repository, and (d) Data Repository–Database. The maximum value for the Coupling Between Objects metric and its upper deviation are considered.

Software metrics Unit Tests and Covered Conditions focus on test implementation and execution in order to determine whether the defined test criteria have been met [17].

In other words, the examined metrics are focused on the dynamic analysis of the software system and their values should be maximized [4]. However, the number of unit tests and conditions to cover are directly related to the specification and implementation of the tested component: in case of a large number of functions and conditions, the number of unit tests and conditions to cover will increase, and vice versa. In that sense, targeted values for Unit Test and Covered Conditions software metrics are defined, as well as their upper and lower deviations.

Software metric Cognitive Complexity focuses on the static analysis of the software system: it considers understandability of the control flow statements within the components [31], which implies that the metric value should be minimized. On the other hand, cognitive complexity is directly related to the implementation of the examined component: in case of a large number of control flow statements, the complexity will be increased, and vice versa. In this context, targeted value for Cognitive Complexity metric and its upper and lower deviations are considered.

Software metrics Security Hotspots and Security Configurations address security issues in a software system. Considering the importance of data found in the software system, software metrics Security Hotspots and Security Configurations should be minimized. Therefore, the maximum values for the examined metrics and their upper deviations are considered.

Deployment Pipelines, AWS Instances, Deployment Profiles, and Healthcheck Endpoints software metrics are focused on the run-time environment of the software system. As the number of instances of the software system increases, the values of the observed metrics also increase. In this context, minimum values for the investigated metrics and their lower deviations are defined.

The notation used to define parameters and variables is as follows:

M—set of features;

Parameters:

k—number of features;

CBO, SH, SC—maximum values of Coupling Between Objects, Security Hotspots, Security Configurations, respectively;

TU, CC, CCM—targeted values of Unit Tests, Covered Conditions, Cognitive Complexity, respectively;

DP, AWS, DPF, HE—minimum values of Deployment Pipelines, AWS Instances, Deployment Profiles, Healthcheck Endpoints, respectively;

Variables:

$$y_j = \begin{cases} 1 & \text{if } j\text{-th feature is microservice or center of monolith} \\ 0 & \text{if } j\text{-th feature is assigned to one of the monoliths} \end{cases}, \quad j \in M$$

$$x_{ij} = \begin{cases} 1 & \text{if } i\text{-th feature is assigned to the } j\text{-th monolith} \\ 0 & \text{otherwise} \end{cases}, \quad i, j \in M$$

n_j—the number of features assigned to the j-th cluster $j \in M$;

$$mo_j = \begin{cases} 1 & \text{if } j\text{-th feature is the center of cluster (monolith)} \\ 0 & \text{otherwise} \end{cases}, \quad j \in M$$

$$ms_j = \begin{cases} 1 & \text{if } j\text{-th feature is microservice} \\ 0 & \text{otherwise} \end{cases}, \quad j \in M$$

$dcbog$, $dtug$, $dccg$, $dccmg$, $dshg$, $dscg$—upper deviations from CBO, TU, CC, CCM, SH, SC, respectively;

dtud, dccd,dccmd,ddpd,dawsd,ddpfd,dhed—lower deviations from TU, CC, CCM, DP, AWS, DPF, HE, respectively.

Using the above notation, the following mixed integer goal programming (MIGP) mathematical model is formulated.

$$\begin{aligned}\min f = \quad & dcbog + dtug + dccg + dccmg + dshg + dscg + \\ & dtud + dccd + dccmd + ddpd + dawsd + ddpfd + dhed\end{aligned}$$

s.t.

$$x_{ij} \leq y_j, \ \ i,j \in M \tag{1}$$

$$x_{ii} = y_j, \ \ i \in M \tag{2}$$

$$\sum_{j \in M} x_{ij} = 1, \ \ i \in M \tag{3}$$

$$\sum_{i \in M} x_{ij} = n_j, \ \ j \in M \tag{4}$$

$$(n_j - 1) - k \cdot mo_j \leq 0, \ \ j \in M \tag{5}$$

$$ms_j = y_j - mo_j, \ \ j \in M \tag{6}$$

$$ms_j \leq n_j, \ \ j \in M \tag{7}$$

$$mo_j \leq n_j, \ \ j \in M \tag{8}$$

$$\sum_{j \in M} (3 \cdot n_j + 1) - \sum_{j \in M} (1 - y_j) - dcbog \leq CBO \tag{9}$$

$$\sum_{j \in M} 18 \cdot n_j + dtud - dtug = TU \tag{10}$$

$$\sum_{j \in M} 6 \cdot n_j + dccd - dccg = CC \tag{11}$$

$$\sum_{j \in M} 3 \cdot n_j + dccmd - dccmg = CCM \tag{12}$$

$$3 \cdot \sum_{j \in M} mo_j + 3 \cdot \sum_{j \in M} ms_j - dshg \leq SH \tag{13}$$

$$\sum_{j \in M} mo_j + \sum_{j \in M} ms_j - dscg \leq SC \tag{14}$$

$$\sum_{j \in M} mo_j + \sum_{j \in M} ms_j + ddpd \geq DP \tag{15}$$

$$\sum_{j \in M} mo_j + \sum_{j \in M} ms_j + dawsd \geq AWS \tag{16}$$

$$3 \cdot \sum_{j \in M} mo_j + 3 \cdot \sum_{j \in M} ms_j + ddpfd \geq DPF \tag{17}$$

$$\sum_{j \in M} mo_j + \sum_{j \in M} ms_j + dhed \geq HE \tag{18}$$

The objective is the sum of all deviations, which should be minimal. The constraints ((1)–(3)) represent standard clustering conditions: the feature can be assigned to the cluster only if the cluster exists ((1) and (2)) and it can be assigned to one cluster exactly (3). Equation (4) is used to count how many features are associated with the cluster and consequently to check the status of the feature: if $n_j \geq 2$ the feature is the centre of a cluster, if $n_j = 1$ the feature is microservice and if $n_j = 0$ the feature is assigned to a cluster (monolith). Auxiliary variables mo_j and ms_j are needed to calculate some of the metrics and are defined using constraints ((5)–(8)). Constraint (9) is related to the maximal acceptable value for the Coupling Between Objects. The second term on the left-hand side of the constraint is introduced to correct the first term, in which the value 1 is summed for all features, even those attached to a monolith. Constraints ((10)–(12))

refer to the targeted values for Unit Tests, Covered Conditions and Cognitive Complexity, respectively. Constraints ((13)–(14)) relate to the maximal acceptable values for Security Hotspots and Security Configurations. The final four constraints ((15)–(18)) refer to the minimal acceptable values for Deployment Pipelines, AWS Instances, Deployment Profiles and Healthcheck Endpoints, respectively.

4.2. The MicroMono Application

In order to apply the developed model, the MicroMono application was designed. The architecture of the MicroMono system is presented in Figure 9, showing components related to user management (i.e., *User Controller*, *User Service*, and *User Repository*), software system definition (i.e., *Software System Controller*, *Software System Service*, and *Software System Repository*), and solution production (i.e., *Solver Controller*, *Solver Service*, and *Solver Repository*). In addition to the graphical user interface, parameters for the Solver component can also be specified via the web service, which means the presented model can also be used in external applications. Monolithic software architecture was applied in the design process of the MicroMono system.

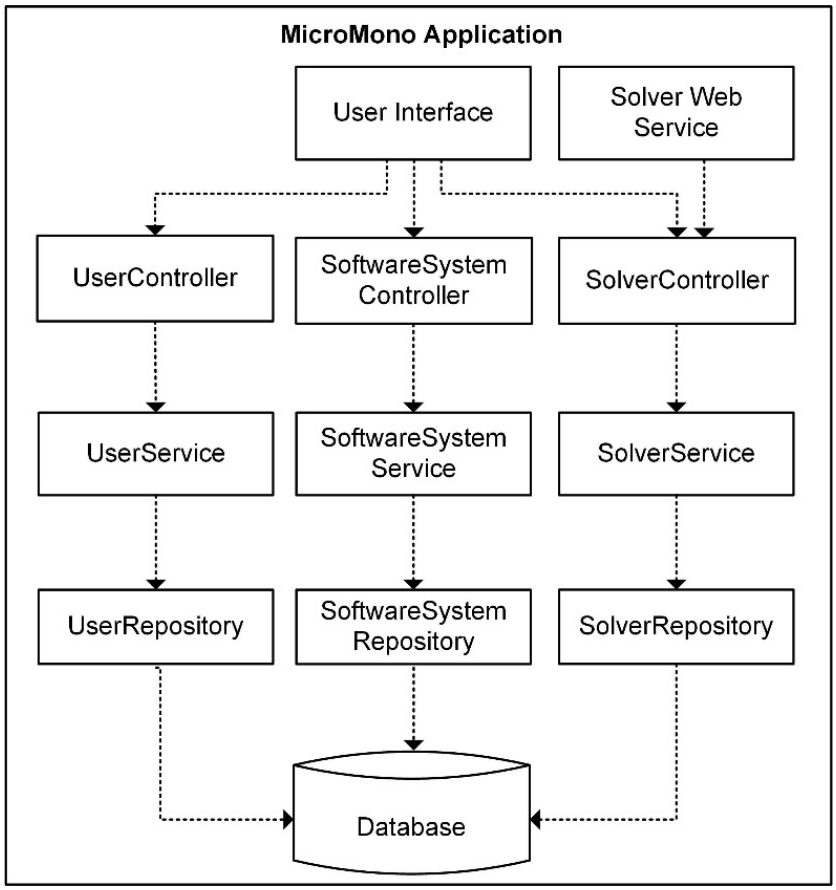

Figure 9. The architecture of the MicroMono application.

After the production of the solution, it is possible to perform a visualization. OrgChart library, which enables the creation of hierarchy diagrams, was applied to this end. The solution can be exported to *Microsoft Excel* for further analysis (see Figure 10).

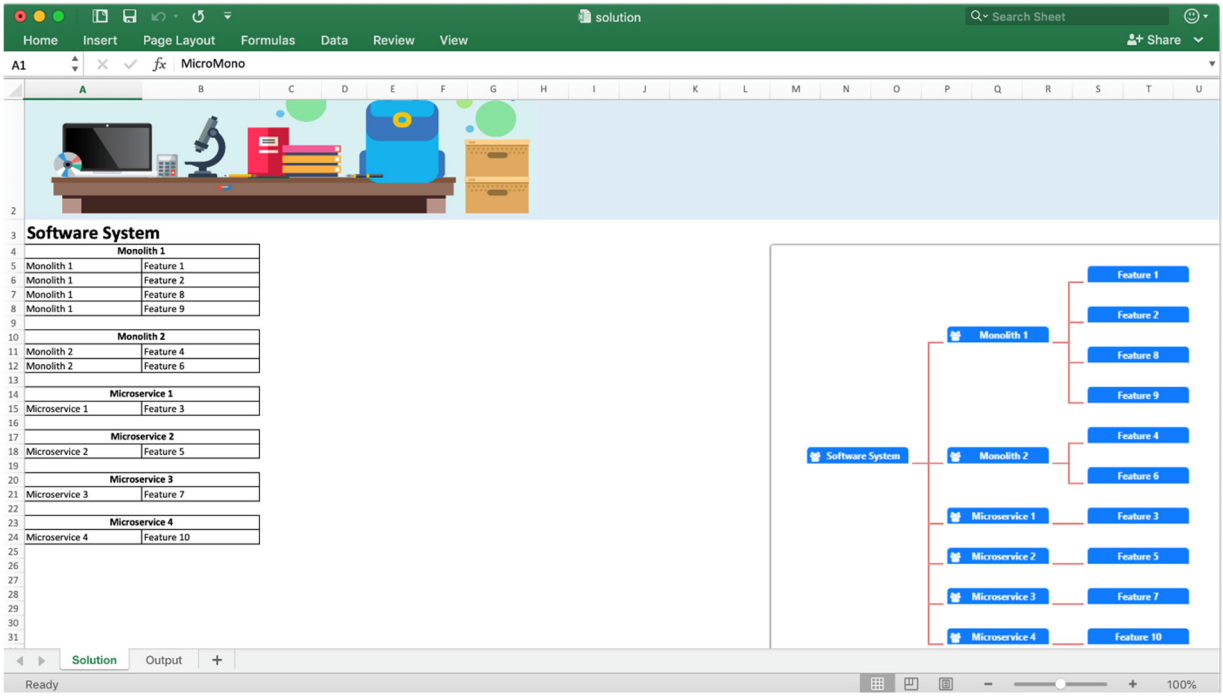

Figure 10. The structure of the solution in textual and graphic form.

5. Threats to Validity

This research examines the general structure of monolithic and microservice architectures. In particular, the research considers monolithic and microservice architecture implementation specific to Jakarta EE and Spring Framework. Although the examined software system contains typical components, implementation techniques may vary depending on the technology platform, applied frameworks, complexity, and domain of the software system [34,35]. A software system may be much more complex and may contain numerous modules related in various ways, which should be considered during the evaluation process. However, the identified modules and components promote separation of concerns between user interface, business logic, and data persistence [30]. In addition, the Single Responsibility Principle is applied [12].

This paper investigates only some of the software quality attributes. However, each quality model can define different quality attributes, i.e., has the ability to decompose quality characteristics and sub-characteristics in diverse ways [17,18]. Additionally, different terms used for the same software quality attribute may exist [23]. For the purpose of obtaining more relevant results, a wider scope of software quality attributes should be considered.

6. Discussion

During the software development process, different aspects (e.g., human resources, equipment, development time, defects and time needed for resolving them) should be examined. These aspects are aimed at creating a software product that brings value to all stakeholders (e.g., software development team, software quality assurance team, project managers, users). Stakeholders can focus on different software development aspects, hence the software development process can be managed by applying a quantitative quality-driven approach [36]. A quantitative approach refers to objective measurement of characteristics, so software quality standards can guide and gather all stakeholders during the software development process, establishing quality-driven software engineering [37,38].

Examination of the presented software architectures implies that principles of decomposition and modularization were applied during the software development process. Decomposition and modularization are crucial and frequently used software design princi-

ples. This helps achieve reusability and extendability of software system components [39]. In other words, the initial problem can be decomposed into simpler sub-problems (i.e., modules) that can be independently realized. Within the monolithic software architecture, the observed modules are dependent upon each other, while the modules in microservice software architecture are independent (i.e., each module represents one microservice). Taking into account a module's independency, microservices communicate with each other by sending messages using synchronous or asynchronous protocols [40].

During the software design process, in accordance with the Single Responsibility Principle [12], each module should have only one responsibility. Put differently, a module should encapsulate only those components that are necessary for the observed responsibility's realization, i.e., the module should be highly cohesive. Moreover, a module's coupling should be decreased to the lowest possible degree, thus achieving the principle "High Cohesion, Low Coupling" [14,41,42].

In microservice software architecture, modules are independent, while within monolithic software architecture, module coupling exists. This coupling can be reduced by defining interfaces for providing communication between modules. As a result, the principle "Program to an interface, not an implementation" is applied [12,43].

In addition to the application of the said principles, microservices also require certain configurations:

- security configuration of each microservice;
- security configuration of communication between microservices;
- external libraries configuration (e.g., database driver, data persistence-related libraries, tests-related libraries);
- configuration of each instance located on cloud infrastructure.

In that regard, we believe that microservice software architecture violates the "Don't Repeat Yourself" principle and promotes the application of the "Write Everything Twice" principle. However, the use of tools for creating the application skeleton (e.g., Maven Archetype, Spring Initializr) can reduce the aforementioned configurations. It is also possible to apply the "Convention over Configuration" principle to minimize the number of decisions a software engineering team needs to make [44]. Nonetheless, these configurations may present a time-consuming task in case of a high number of microservices.

Software quality standards define quality models and attributes [17,18]. Software quality attributes refer to non-functional characteristics of a software system [24], so it is possible to establish a relationship between software architectures and software quality attributes [23]. Fazio et al. examined monolithic and microservice software architectures in their research and concluded that the implementation of microservice architecture positively impacts scalability, portability, updatability, and availability quality attributes, but at the cost of expensive remote calls (instead of in-process calls) and increased overheads for cross-component synchronization [13]. Furthermore, microservice architecture implementation may also have a positive effect on testability, reliability, and maintainability quality attributes [11]. Each microservice can be independently scalable, portable, updatable, available, testable, reliable, and maintainable. In addition, usage of microservices can have a beneficial effect on deployability, modifiability, and resilience quality attributes, which results in continuous delivery and DevOps software development [25,26]. In that respect, microservices can be distributed within the container that can be executed on various cloud vendor's infrastructures [45]. Contrary to monolithic applications, the use of microservices may also result in an increase of both team's agility and scalability. In addition, tech stack for each microservice can be considered, which results in polyglot software development [34,46].

Implementation of microservice architecture helps achieve isolation of a software system's module. However, this exact isolation as well as numerous microservices may present a great challenge to the software development process management [11,25]. On the other hand, by comparing monolithic and microservice software architectures, it can be concluded that the implementation of microservice architecture may have a negative impact

on security, performance, and testability quality attributes [11]. Due to a microservice's independence, data transfer (e.g., for encryption and authorization in communication between microservices) requires additional effort in providing security. Communication over a network requires managing a network's infrastructure and may contribute to additional latency that should be taken into consideration [47].

It was previously mentioned that each microservice could be independently tested. However, integration testing of a system that implements a microservice software architecture may be rather complicated [25]. On the other hand, modules within the monolithic software architecture present an entity, so their integration testing is simpler. In addition, modules in monolithic software architecture can be tested in isolation if interfaces for communication between components are properly defined, which achieve the principle "Separation of interface and implementation" [48]. The abovementioned indicates that monolithic and microservice architectures can be examined in the context of different quality attributes.

Monolithic and microservice architectures can be observed in the context of design pattern. Design pattern could be defined as a three-part rule that expresses the relationship between a certain problem, its solution and their context [43]. In that regard, various design patterns can be implemented in the software development process [43,49,50], which can also have an impact on software quality [51,52].

The presented quality-based model considers monolith and microservice software architectures in the context of software metrics and software system features. In order to illustrate the formulated mathematical model, a set of experiments was performed. The dimensions of the solved problems and the optimization results are shown in Table 4. The models were solved for software systems with k = 10, 20, 30, 40, 50, 100, 150, 200, 250 and 300 features. The right-hand side values of constraints ((9)–(18)) were determined as follows: first, for each k using equations from Table 3, the values of software metrics are determined for two extreme cases: all features are contained in one monolith (met_1) and each feature is one microservice (met_2). Then, the obtained values were aggregated into right-hand side values (rhs) using the equation:

$$rhs = w_1 \cdot met_1 + w_2 \cdot met_2 \tag{19}$$

The weights of w_1 and w_2 were incrementally changed and their values are shown in the header of Table 4.

The elements of the solutions for each k and each weights ratio are given in Table 4: the number of monoliths with the number of features given in parentheses; the number of microservices; the sum of deviations (the value of the objective) and the optimization time. Optimization was performed on Lenovo ThinkPad L580 Laptop (Intel Core 8th Generation i7-8550U at 2.00 GHz Processor, 16 GB DDR4 RAM, 512 GB SSD, Windows 10 Professional 64-bit Operating System).

As expected, with an increase in the weight of met_1 and met_2, the number of features organized into microservices increased, i.e., the number of features grouped into monoliths decreased. In addition, in most cases, features were grouped into multiple monoliths of different sizes. In 20 cases, the value of the objective is 0, i.e., there is no deviation from the maximal, minimal and target values of the metrics. When weights w_1 and w_2 are equal or close, the objective function value is equal to 1. In all 30 such cases, deviations from maximal values of Security Hotspots (SH) by 1 are obtained. Given that the values for SH are high, for example SH = 182 for k = 100, the obtained deviations are acceptable. In the last 20 cases, the objective values are equal to 2 and they all refer to the deviation from maximal values of Security Hotspots. Based on the performed examinations, the internal structure of the software is defined in a consistent manner, while the external software behaviour remains unchanged.

Table 4. Optimization results.

No. of Features	Solution	$w_1 = 0.8$ $w_2 = 0.2$	$w_1 = 0.7$ $w_2 = 0.3$	$w_1 = 0.6$ $w_2 = 0.4$	$w_1 = 0.5$ $w_2 = 0.5$	$w_1 = 0.4$ $w_2 = 0.6$	$w_1 = 0.3$ $w_2 = 0.7$	$w_1 = 0.2$ $w_2 = 0.8$
10	# of monoliths	1 (8)	1 (7)	1 (6)	2 (4, 2)	1 (4)	1 (3)	1 (2)
	# of microservices	2	3	4	4	6	7	8
	deviation	0	0	1	1	1	2	2
	time used	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s
20	# of monoliths	1 (16)	1 (14)	1 (12)	2 (8, 3)	1 (8)	2 (5, 2)	1 (4)
	# of microservices	4	6	8	9	12	13	16
	deviation	0	0	1	1	1	2	2
	time used	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s	0.0 s
30	# of monoliths	1 (24)	1 (21)	1 (18)	1 (15)	3 (6, 6, 2)	1 (9)	1 (6)
	# of microservices	6	9	12	15	16	21	24
	deviation	0	0	1	1	1	2	2
	time used	0.0 s	0.0 s	0.0 s	0.0 s	0.1 s	0.0 s	0.1 s
40	# of monoliths	1 (32)	1 (28)	1 (24)	1 (20)	1 (16)	2 (10, 3)	3 (4, 3, 3)
	# of microservices	8	12	16	20	24	27	30
	deviation	0	0	1	1	1	2	2
	time used	0.0 s	0.0 s	0.0 s	0.0 s	0.2 s	0.1 s	0.2 s
50	# of monoliths	1 (40)	1 (35)	1 (30)	4 (22, 2, 2, 2)	3 (15, 5, 2)	1 (15)	2 (8, 3)
	# of microservices	10	15	20	22	28	35	39
	deviation	0	0	1	1	1	2	2
	time used	0.0 s	0.0 s	0.0 s	0.3 s	0.8 s	0.3 s	0.3 s
100	# of monoliths	1 (80)	1 (70)	1 (60)	5 (29, 4, 4, 2, 15)	3 (29, 8, 5)	3 (14, 9, 9)	2 (12, 9)
	# of microservices	20	30	40	46	58	68	79
	deviation	0	0	1	1	1	2	2
	time used	0.1 s	0.2 s	0.2 s	2.5 s	2.2 s	0.8 s	0.7 s
150	# of monoliths	1 (120)	1 (105)	1 (90)	2 (62, 14)	2 (58, 3)	6 (36, 6, 2, 2, 2, 2)	4 (21, 2, 7, 3)
	# of microservices	30	45	60	74	89	100	117
	deviation	0	0	1	1	1	2	2
	time used	0.4 s	0.3 s	0.4 s	3.4 s	18.8 s	2.0 s	40.8 s
200	# of monoliths	1 (160)	1 (140)	1 (120)	7 (49, 8, 37, 2, 3, 4, 3)	1 (80)	3 (45, 15, 2)	4 (28, 11, 2, 2)
	# of microservices	40	60	80	94	120	138	157
	deviation	0	0	1	1	1	2	2
	time used	0.6 s	0.6 s	0.7 s	33.2 s	15.9 s	23.2 s	17.3 s
250	# of monoliths	1 (200)	1 (175)	1 (150)	18 *	35 **	1 (75)	15 ***
	# of microservices	50	75	100	108	116	175	186
	deviation	0	0	1	1	1	2	2
	time used	0.9 s	0.9 s	0.9 s	198.4 s	161.9 s	22.7 s	74.6 s
300	# of monoliths	1 (240)	1 (210)	1 (120)	8 (87, 38, 10, 9, 3, 6, 2, 2)	5 (76, 17, 22, 6, 3)	6 (21, 50, 4, 12, 4, 4)	3 (42, 18, 2)
	# of microservices	60	90	180	143	176	205	238
	deviation	0	0	1	1	1	2	2
	time used	1.3 s	1.3 s	1.3 s	41.0 s	54.5 s	87.5 s	37.4 s

* 18 (79, 3, 3, 2, 22, 2, 2, 2, 6, 4, 2, 2, 3, 2, 2, 2, 2, 2) ** 35 (7, 2, 2, 2, 2, 4, 5, 3, 3, 3, 3, 3, 3, 4, 3, 3, 2, 5, 3, 3, 3, 3, 3, 2, 4, 8, 6, 5, 8, 2, 2, 4, 3, 9, 7) *** 15 (15, 3, 2, 5, 6, 2, 2, 2, 2, 10, 2, 2, 4, 4, 3).

Benefits of software architectures obtained by optimization compared to only monolith or only microservices architectures are illustrated through a comparison of software metric values for different numbers of features from Table 4. Due to the large number of instances, those with weight values $w_1 = w_2 = 0.5$ were selected and shown in Table 5. A related type of constraint (9–18) for each of the 10 software metrics is shown in the mathematical model in the "constraint type" row, while the right sizes of those constraints for $k \in \{10, 20, 30, 40, 50, 100, 150, 200, 250, 300\}$ are given in the "condition" rows. The rows "monolith", "microservice", and "optimal" contain the deviations from the given conditions for software metric values for the cases of only monolith, only microservices and optimal software architectures, respectively.

In the software arhitecture obtained by optimization, the only deviation appears for Security Hotspots (SH), where the value of this metric is obtained by 1 higher than the maximal required. In the case of a monolithic architecture, none of the metrics with minimum acceptable values (Deployment Pipelines, AWS Instances, Deployment Profiles, Healthcheck Endpoints) are satisfied. For example, the value of Deployment Profiles (DP) for $k = 200$ according to the constraint (17) should be at least 302, but in the monolith arhitecture its value is 3, i.e., deviation is 299. In the case of microservice arhitecture, the metrics with maximum acceptable values (Coupling Between Objects, Security Hotspots,

Security Configurations) are not satisfied. For example, the value of Security Configurations (SC) for k = 200 according to the constraint (14) can be less than or equal to 101, but in microservice arhitecture its value is 200, i.e., deviation is 99. For both architectures, monolith and microservice, the higher the number of features, the greater these deviations. In some other ratios of weights w_1 and w_2, the deviations are even larger.

Table 5. Deviation comparison.

		Software Metric									
		CBO	**TU**	**CC**	**CCM**	**SH**	**SC**	**DP**	**AWS**	**DPF**	**HE**
k	constraint type	<	=	=	=	<	<	>	>	>	>
10	condition	36	180	60	30	17	6	6	6	17	6
	monolith	-	-	-	-	-	-	5	5	14	5
	microservice	4	-	-	-	13	4	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
20	condition	71	360	120	60	32	11	11	11	32	11
	monolith	-	-	-	-	-	-	10	10	29	10
	microservice	9	-	-	-	28	9	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
30	condition	106	540	180	90	47	16	16	16	47	16
	monolith	-	-	-	-	-	-	15	15	44	15
	microservice	14	-	-	-	43	14	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
40	condition	141	720	240	120	62	21	21	21	62	21
	monolith	-	-	-	-	-	-	20	20	59	20
	microservice	19	-	-	-	58	19	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
50	condition	176	900	300	150	77	26	26	26	77	26
	monolith	-	-	-	-	-	-	25	25	74	25
	microservice	24	-	-	-	73	24	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
100	condition	351	1800	600	300	152	51	51	51	152	51
	monolith	-	-	-	-	-	-	50	50	149	50
	microservice	49	-	-	-	148	49	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
150	condition	526	2700	900	450	227	76	76	76	227	76
	monolith	-	-	-	-	-	-	75	75	224	75
	microservice	74	-	-	-	223	74	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
200	condition	701	3600	1200	600	302	101	101	101	302	101
	monolith	-	-	-	-	-	-	100	100	299	100
	microservice	99	-	-	-	298	99	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
250	condition	876	4500	1500	750	377	126	126	126	377	126
	monolith	-	-	-	-	-	-	125	125	374	125
	microservice	124	-	-	-	373	124	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-
300	condition	1051	5400	1800	900	452	151	151	151	452	151
	monolith	-	-	-	-	-	-	150	150	449	150
	microservice	149	-	-	-	448	149	-	-	-	-
	optimal	-	-	-	-	1	-	-	-	-	-

Furthermore, let us examine software metrics values in case of one feature. Under these circumstances, the following software metrics values are achieved:

$$Coupling\ Between\ Objects_{Monolithic} = Coupling\ Between\ Objects_{Microservice} = 4,\ k = 1 \quad (20)$$

$$Unit\ Tests_{Monolithic} = Unit\ Tests_{Microservice} = 18,\ k = 1 \quad (21)$$

$$Covered\ Conditions_{Monolithic} = Covered\ Conditions_{Microservice} = 6,\ k = 1 \quad (22)$$

$$Cognitive\ Complexity_{Monolithic} = Cognitive\ Complexity_{Microservice} = 3,\ k = 1 \quad (23)$$

$$Security\ Hotspots_{Monolithic} = Security\ Hotspots_{Microservice} = 3,\ k = 1 \quad (24)$$

$$Security\ Configurations_{Monolithic} = Security\ Configurations_{Microservice} = 1,\ k = 1 \quad (25)$$

$$Deployment\ Pipelines_{Monolithic} = Deployment\ Pipelines_{Microservice} = 1,\ k = 1 \quad (26)$$

$$AWS\ Instances_{Monolithic} = AWS\ Instances_{Microservice} = 1,\ k = 1 \quad (27)$$

$$Deployment\ Profiles_{Monolithic} = Deployment\ Profiles_{Microservice} = 3,\ k = 1 \quad (28)$$

$$Healthcheck\ Endpoints_{Monolithic} = Healthcheck\ Endpoints_{Microservice} = 1,\ k = 1 \quad (29)$$

It is evident that, in this case, the observed software quality metrics will have the same values in both monolithic and microservice software architectures. In case of only one feature, it may be concluded that the point $k = 1$ represents the *Intersection Point of Monolithic and Microservice Software Architectures* where the examined quality metrics obtain the same values. On the other hand, the increase in the number of features leads to a value increase of quality metrics in microservice software architecture, whilst these values are constant in monolithic software architecture (with the exception of Coupling Between Objects, Units Tests, Covered Conditions, and Cognitive Complexity metrics with values increasing in both architectures). Against this background, two conclusions may be drawn:

- In case of only one feature (i.e., $k = 1$) monolithic or microservice software architectures could be applied either way;
- In case of multiple features (i.e., $k > 1$), software quality attribute's importance should be examined and a decision on whether to implement monolithic or microservice software architecture should be made.

Taking into account the presented discussion, practical quality-based recommendations related to monolith and microservice architectures can be introduced. When Testability, Coupling and Complexity are concerned, the evaluation indicates that each microservice encapsulates its tests, coupling and complexity. Lower code coverage (in terms of Testability), coupling between objects (in terms of Coupling), as well as cyclomatic complexity and cognitive complexity (in terms of Complexity) were obtained in microservice software architecture. Therefore, if the number of features increases, the complexity could be divided between microservices. Additionally, when increasing the number of features, unit testing of the microservice is simple, while integration testing may pose challenges.

On the other hand, some quality metrics in case of an increase in the number of features indicate that the application of microservice software architecture may be a more preferable solution. The corresponding software system can be more deployable and available compared to a monolithic system. As for Deployability metrics, multiple AWS instances, pipelines, and deployment profiles exist. In that context, if a feature changes, microservice's deployable can be deployed faster than that of a monolithic application's. Furthermore, parallel execution of microservices' pipelines is possible, which additionally reduces the deployment time. As for Availability metrics, each microservice can independently be healthchecked based on which a new microservice instance can be started. Furthermore, when high-availability of a service is needed, auto-provisioning and scaling could be performed. When Security metrics are concerned, an increase in the number of features will not affect security hotspots and security configuration of monolithic application, while the security of each microservice should particularly be assessed. Considering that an increase

in the number of features implies that each microservice as well as their communication need to be more secured, along with the fact there is a need for managing a network's infrastructure, performances and latency, monolithic application can represent a better solution. The above conclusions may contribute to the appropriate choice of architecture that best corresponds to the software system being developed.

Different research focused on choosing a technique for microservice's extraction from a monolithic system can be found. Research conducted by Levcovitz et al. defines a set of facades, set of business functions and set of database tables [53]. Subsequently, component mappings are defined, dependency graph is created, and microservice candidates are identified [53]. Within the examined software system in this approach, controllers correspond to facades, business functions could be presented through the services, while each microservice would incorporate a relevant repository component and relevant database table. Another approach suggests microservice organization around business capabilities [11,45]. According to this approach, a set of similar functionalities should be grouped into one microservice. Within the examined software system, a microservice could be presented using one feature which can be defined as a functional characteristic of a system of interest that end-users and other stakeholders can understand [28]. This helps identify business capabilities that could be implemented as microservices. The following approach implies the application of Domain-Driven Design techniques and Bounded Context pattern [54], which helps achieve modelling according to domains. When applying this approach to the examined software system, microservices could be modelled towards domain objects which would result in bounded context: each microservice is aimed at one domain and incorporates only domain-related functionalities. Although the resulting microservices correspond to the observed microservice extraction techniques, we do believe the results may vary depending on the domain and complexity of the system subject to evaluation.

Each software architecture has certain benefits. Microservice architecture is a preferable option when many of the quality attributes are considered, but monolithic architecture has its benefits as well. Within a software development industry, numerous legacy systems that usually use monolithic architecture exist [14,55]. Software requirements ask for continuous improvement and refinement of architecture [6]. So, a transition from monolithic to microservice architecture is possible with the application of *Strangler* design pattern by transforming each functionality to a separate microservice [56,57]. This accomplishes refactoring, i.e., the system's design is changed without changing external software behavior [58,59]. However, it is important to note that refactoring of the whole system is not conducted at once, but isolation of functionalities and their transformation to microservices is achieved gradually. Refactoring should consequently lead to the improvement of software system's quality level [60,61].

Finally, we want to emphasize that the application of microservice software architecture will not resolve all problems in the software development process [14]. Implementation of monolithic or microservice architecture may have a positive impact on some software quality attributes, while at the same time having a negative effect on other attributes [11,62]. Therefore, quality attributes cannot be observed in isolation, but rather as a part of the whole development process.

7. Conclusions

The complexity of modern software systems enforces the need for good organization of the software development process. One of the important decisions the software development team needs to make is the selection of software architecture which is the basis for further software development. This paper has discussed monolithic and microservice software architectures. The evaluation process examined the software system for project assessment, while Feature-driven development and Jakarta EE technology stack were applied in the software development process.

Based on the performed evaluation, a quality-based mathematical model for software architecture optimization was developed. The MIGP model incorporates various software

metrics in terms of their values and deviations and produces a solution which incorporates multiple monoliths and microservices. In addition, an intersection point in which observed software metrics obtain the same values in monolithic and microservice software architectures was introduced. This is achieved only in the case of one feature and implies that either one of the architectures, monolithic or microservice, can be applied. On the other hand, an increase in the number of features indicates that the importance of quality attribute for the software system should examined and, in accordance with that, software architecture should be selected.

Furthermore, practical recommendations regarding software architectures, in the context of software quality, were presented. When Testability, Coupling, and Complexity are concerned, research indicates that each microservice encapsulates its tests, coupling and complexity. Therefore, in case of an increase in the number of features, the complexity could be divided between microservices. Additionally, when increasing the number of features, unit testing of the microservice is simple, while integration testing may pose challenges. On the other hand, Deployability and Availability quality metrics obtain better values in case of microservice architecture, while Security quality metrics have more preferable values within monolithic architecture. Taking into account all these conclusions can help make a proper choice of architecture that best suits the specific software system.

Since each software system, apart from functional requirements, also needs to meet non-functional requirements related to software quality [4,24], we believe that the obtained conclusions form a good basis for further research. Provided that this research had discussed the general structure of monolithic and microservice software architectures, further direction of examination may refer to the inclusion of additional criteria in the evaluation process (e.g., additional coupling between components and/or modules, examination of interdependence between quality attributes). Although the investigated features of the software system contain typical components and operations, further research should include additional components and operations. In addition, the importance of software quality attributes for the software system being developed should be examined. Taking into account that each quality model can define different quality attributes, further research should also include a larger set of software quality attributes.

Considering the direct relation between software architectures and quality attributes, software development that focuses on quality is promoted in this way and, consequently, the improvement of software system's quality is achieved. This helps establish quality-driven software engineering [37,38].

Author Contributions: Conceptualization, M.M. and D.M.-N.; methodology, M.M. and D.M.-N.; software, M.M.; validation, M.M. and D.M.-N.; investigation, M.M. and D.M.-N.; data curation, M.M. and D.M.-N.; writing—original draft preparation, M.M. and D.M.-N.; writing—review and editing, M.M. and D.M.-N.; visualization, M.M.; supervision, M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pressman, R.S.; Maxim, B.R. *Software Engineering: A Practitioner's Approach*, 9th ed.; McGraw-Hill Education: New York, NY, USA, 2019.
2. Sommerville, I. *Software Engineering*, 9th ed.; Addison-Wesley Publishing: Boston, MA, USA, 2015.
3. Kumar, G.; Bhatia, P.K. Comparative analysis of software engineering models from traditional to modern methodologies. In Proceedings of the 2014 Fourth International Conference on Advanced Computing & Communication Technologies, Rohtak, India, 8–9 February 2014; pp. 189–196.
4. Bourque, P.; Fairley, R.E. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*; IEEE Computer Society Press: Piscataway, NJ, USA, 2014.

5. Medvidovic, N.; Taylor, R.N. Software architecture: Foundations, theory, and practice. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, Cape Town, South Africa, 2–8 May 2010; pp. 471–472.
6. Medvidovic, N.; Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.* **2000**, *26*, 70–93. [CrossRef]
7. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*; Addison-Wesley Professional: Boston, MA, USA, 2003.
8. Chen, H.-M.; Kazman, R.; Perry, O. From Software Architecture Analysis to Service Engineering: An Empirical Study of Methodology Development for Enterprise SOA Implementation. *IEEE Trans. Serv. Comput.* **2010**, *3*, 145–160. [CrossRef]
9. ISO/IEC/IEEE 24765:2017 Systems and Software Engineering—Vocabulary. Available online: http://www.iso.org (accessed on 31 July 2022).
10. Clements, P.; Garlan, D.; Bass, L.; Stafford, J.; Nord, R.; Ivers, J.; Little, R. *Documenting Software Architectures: Views and Beyond*; Pearson Education: London, UK, 2002.
11. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*; Mazzara, M., Meyer, B., Eds.; Springer: Cham, Switzerland, 2017; pp. 195–216.
12. Martin, R.C.; Martin, M. *Agile Principles, Patterns, and Practices in C#*; Prentice Hall: Hoboken, NJ, USA, 2006.
13. Fazio, M.; Celesti, A.; Ranjan, R.; Liu, C.; Chen, L.; Villari, M. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Comput.* **2016**, *3*, 81–88. [CrossRef]
14. Newman, S. *Building Microservices: Designing Fine-Grained Systems*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015.
15. Bansiya, J.; Davis, C.G. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.* **2002**, *28*, 4–17. [CrossRef]
16. Crosby, P.B. *Quality is Free*; Signet Book: New York, NY, USA, 1980.
17. ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. Available online: http://www.iso.org (accessed on 31 July 2022).
18. ISO/IEC 9126:2001 Software Engineering—Product Quality—Part 1: Quality Model. Available online: http://www.iso.org (accessed on 31 July 2022).
19. Franch, X.; Carvallo, J. Using quality models in software package selection. *IEEE Softw.* **2003**, *20*, 34–41. [CrossRef]
20. Suresh, Y.; Pati, J.; Rath, S.K. Effectiveness of Software Metrics for Object-oriented System. *Procedia Technol.* **2012**, *6*, 420–427. [CrossRef]
21. Kan, S.H. *Metrics and Models in Software Quality Engineering*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
22. Balogun, A.; Basri, S.; Mahamad, S.; Abdulkadir, S.; Almomani, M.; Adeyemo, V.; Al-Tashi, Q.; Mojeed, H.; Imam, A.; Bajeh, A. Impact of Feature Selection Methods on the Predictive Performance of Software Defect Prediction Models: An Extensive Empirical Study. *Symmetry* **2020**, *12*, 1147. [CrossRef]
23. Alshuqayran, N.; Ali, N.; Evans, R. A systematic mapping study in microservice architecture. In Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), Macau, China, 4–6 November 2016; pp. 44–51.
24. Chung, L.; do Prado Leite, J.C.S. On non-functional requirements in software engineering. In *Conceptual Modeling: Foundations and Applications*; Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 363–379.
25. Chen, L. Microservices: Architecting for continuous delivery and DevOps. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018; pp. 39–397.
26. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Softw.* **2016**, *33*, 42–52. [CrossRef]
27. Palmer, S.R.; Felsing, M. *A Practical Guide to Feature-Driven Development*; Pearson Education: London, UK, 2001.
28. ISO/IEC 26550:2015 Software and Systems Engineering—Reference Model for Product Line Engineering and Management. Available online: http://www.iso.org (accessed on 31 July 2022).
29. Taibi, D.; Lenarduzzi, V.; Pahl, C. Architectural Patterns for Microservices: A Systematic Mapping Study. In Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER), Funchal, Portugal, 19–21 March 2018; pp. 221–232.
30. Hasselbring, W.; Reussner, R.; Jaekel, H.; Schlegelmilch, J.; Teschke, T.; Krieghoff, S. The dublo architecture pattern for smooth migration of business information systems: An experience report. In Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE), Edinburgh, UK, 28–28 May 2004; pp. 117–126.
31. Campbell, G.A. Cognitive complexity: An overview and evaluation. In Proceedings of the 2018 International Conference on Technical Debt, Gothenburg, Sweden, 27–28 May 2018; pp. 57–58.
32. Atkinson, C.; Bostan, P.; Hummel, O.; Stoll, D. A practical approach to web service discovery and retrieval. In Proceedings of the IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, UT, USA, 9–13 July 2007; pp. 241–248.
33. Cardarelli, M.; Iovino, L.; Di Francesco, P.; Di Salle, A.; Malavolta, I.; Lago, P. An extensible data-driven approach for evaluating the quality of microservice architectures. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; pp. 1225–1234.
34. Zimmermann, O. Microservices tenets. *Comput. Sci. Res. Dev.* **2017**, *32*, 301–310. [CrossRef]
35. Calderón-Gómez, H.; Mendoza-Pittí, L.; Vargas-Lombardo, M.; Gómez-Pulido, J.; Rodríguez-Puyol, D.; Sención, G.; Polo-Luque, M.-L. Evaluating Service-Oriented and Microservice Architecture Patterns to Deploy eHealth Applications in Cloud Computing Environment. *Appl. Sci.* **2021**, *11*, 4350. [CrossRef]

36. Al-Naeem, T.; Gorton, I.; Babar, M.A.; Rabhi, F.; Benatallah, B. A quality-driven systematic approach for architecting distributed software applications. In Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 15–21 May 2005; pp. 244–253.
37. Tahvildari, L.; Kontogiannis, K.; Mylopoulos, J. Quality-driven software re-engineering. *J. Syst. Softw.* **2003**, *66*, 225–239. [CrossRef]
38. Villegas, N.M.; Müller, H.A.; Tamura, G.; Duchien, L.; Casallas, R. A framework for evaluating quality-driven self-adaptive software systems. In Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Honolulu, HI, USA, 23–24 May 2011; pp. 80–89.
39. Mayer, T.; Hall, T. A Critical Analysis of Current OO Design Metrics. *Softw. Qual. J.* **1999**, *8*, 97–110. [CrossRef]
40. Jamshidi, P.; Pahl, C.; Mendonca, N.C.; Lewis, J.; Tilkov, S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Softw.* **2018**, *35*, 24–35. [CrossRef]
41. Hitz, M.; Montazeri, B. Measuring coupling and cohesion in object-oriented systems. In Proceedings of the International Symposium on Applied Corporate Computing (ISACC), Cairns, Australia, 4–6 December 1995; pp. 1–10.
42. Jin, W.; Liu, T.; Qu, Y.; Zheng, Q.; Cui, D.; Chi, J. Dynamic structure measurement for distributed software. *Softw. Qual. J.* **2017**, *26*, 1119–1145. [CrossRef]
43. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995.
44. Bächle, M.; Kirchberg, P. Ruby on rails. *IEEE Softw.* **2007**, *24*, 105–108. [CrossRef]
45. Pahl, C.; Jamshidi, P. Microservices: A Systematic Mapping Study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016), Rome, Italy, 23–25 April 2016; Volume 1, pp. 137–146.
46. Hasselbring, W.; Steinacker, G. Microservice architectures for scalability, agility and reliability in e-commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 243–246.
47. Villamizar, M.; Garcés, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Casallas, R.; Gil, S.; Valencia, C.; Zambrano, A.; et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Serv. Oriented Comput. Appl.* **2017**, *11*, 233–247. [CrossRef]
48. Crnkovic, I.; Hnich, B.; Jonsson, T.; Kiziltan, Z. Specification, implementation, and deployment of components. *Commun. ACM* **2002**, *45*, 35–40. [CrossRef]
49. Burns, B.; Oppenheimer, D. Design patterns for container-based distributed systems. In Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO, USA, 20–21 June 2016.
50. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
51. Ampatzoglou, A.; Frantzeskou, G.; Stamelos, I. A methodology to assess the impact of design patterns on software quality. *Inf. Softw. Technol.* **2012**, *54*, 331–346. [CrossRef]
52. Khomh, F.; Gueheneuce, Y.G. Do design patterns impact software quality positively? In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, 1–4 April 2008; pp. 274–278.
53. Levcovitz, A.; Terra, R.; Valente, M.T. Towards a technique for extracting microservices from monolithic enterprise systems. In Proceedings of the 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), Belo Horizonte, Brazil, 23 September 2015; pp. 97–104.
54. Carrasco, A.; Bladel, B.V.; Demeyer, S. Migrating towards microservices: Migration and architecture smells. In Proceedings of the 2nd International Workshop on Refactoring, Montpellier, France, 4 September 2018; pp. 1–6.
55. Sneed, H.M. Integrating legacy software into a service oriented architecture. In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06), Bari, Italy, 22–24 March 2006; pp. 11–14.
56. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann, A. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 187–195.
57. Taibi, D.; Lenarduzzi, V.; Pahl, C. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Comput.* **2017**, *4*, 22–32. [CrossRef]
58. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 2018.
59. Mens, T.; Tourwé, T. A survey of software refactoring. *IEEE Trans. Softw. Eng.* **2004**, *30*, 126–139. [CrossRef]
60. Alshayeb, M. Empirical investigation of refactoring effect on software quality. *Inf. Softw. Technol.* **2009**, *51*, 1319–1326. [CrossRef]
61. Fontana, F.A.; Spinelli, S. Impact of refactoring on quality code evaluation. In Proceedings of the 4th Workshop on Refactoring Tools, Honolulu, HI, USA, 22 May 2011; pp. 37–40.
62. Hassan, S.; Bahsoon, R. Microservices and their design trade-offs: A self-adaptive roadmap. In Proceedings of the 2016 IEEE International Conference on Services Computing (SCC), San Francisco, CA, USA, 27 June–2 July 2016; pp. 813–818.