Computer Science and Information Systems 23(1):1–31 https://doi.org/10.2298/CSIS251110001M

Optimizing Cell-Based Software Architecture through Heuristic Community Detection Approach*

Miloš Milić, Nebojša Nikolić, and Dragana Makajić-Nikolić

University of Belgrade - Faculty of Organizational Sciences
Jove Ilića 154, Belgrade
milos.milic@fon.bg.ac.rs
nebojsa.nikolic@fon.bg.ac.rs
dragana.makajic-nikolic@fon.bg.ac.rs

Abstract. The goal of this research is to investigate the optimization of the Cell-based software architecture. Cell-based software architecture structures a software system into interconnected cells, each comprising multiple elements. This study focuses on optimizing the architecture by determining the optimal number of cells and their internal organization. To achieve this, the Community Detection approach, which identifies closely connected elements, was applied. To preserve cell boundaries, reduce complexity, and enhance modularity, we introduce the concept of functionality, which can be represented by one or more cells. This concept serves as the foundation for optimizing software architecture. A series of experiments were conducted to analyze the problem dimensions that can be addressed through optimization and to evaluate the robustness of the mathematical model. Given that the proposed model is unable to solve large-scale problems efficiently, we developed a heuristic approach and compared its results with those obtained from the mathematical model. The evaluation results indicate that different software architectures can be derived in terms of cell granularity, composition, and interaction. Since each cell can contain multiple elements realized in various architectural styles, the proposed model enables the integration of diverse architectures within a single software system. This flexibility enhances the system's adaptability and overall efficiency.

Keywords: software architecture, cell-based architecture, community detection, architecture optimization, destroy and repair.

1. Introduction

In today's digital age, the application of software systems spans across various domains. These systems enable seamless communication and data exchange within and across different industries. In the interconnected world, software systems can be utilized by a diverse range of clients, and it is essential to ensure they have capabilities to support them effectively.

Software architecture of a system can be defined as the set of elements needed to reason about the system [6], encompassing various software components, their relationships, as well as the properties of components and relationships [12]. Software architecture can

* The paper is an extended version of the FedCSIS 2024 conference paper.

be considered as a blueprint for further software design, based on which various components are created. These software components are used to fulfill the functional requirements of the software system [12].

In addition to functional requirements, software systems should incorporate capabilities related to non-functional requirements such as security, deployability, availability, scalability, reliability, resilience, maintainability, etc. [12]. However, achieving a high level of non-functional requirements can be a challenging task. Non-functional requirements are typically defined as quality attributes of a software system, and are closely related to software architecture [14].

Considering utilization of software systems in various domains, as well as their complexity, ensuring high-quality software becomes an important challenge. Poor architectural decisions can lead to issues such as technical debt, inappropriate resource utilization, and security vulnerabilities [11], [13]. In addition, these decisions are essential in determining the maintainability of software systems, as they establish the foundational structure that influences future software development and evolution [61]. Inappropriate architectural choices can lead to the accumulation of Architectural Technical Debt (ATD) [8], [62], which refers to design decisions that expedite short-term development but hinder long-term system evolution and maintenance [8]. This debt manifests through increased complexity and reduced system adaptability, resulting in high maintenance costs and efforts [8], [62].

Understanding the relationship between software architecture and software quality is essential for developing robust, scalable, secure, and maintainable systems. In this context, software architects and engineers should consider these aspects from the earliest stages of software development [25].

This research investigates the optimization of the Cell-based software architecture. Cell-based architecture considers the organization of a software system in the form of interconnected cells, while each cell can contain multiple elements [1], [55]. The Cell-based software architecture allows each software element within a cell to be realized using its own architectural style, enabling flexibility in design and implementation [1], [55]. By leveraging this approach, the architecture can utilize the benefits of multiple architectural paradigms within different cells. Since each cell operates autonomously and can be managed independently of others, this architecture enhances encapsulation, isolation, and the distribution of software elements [1], [55]. Some open topics in this approach relate to determining the optimal granularity, composition, and interaction of cells [52], as these factors directly impact various software quality aspects. This research examines the optimization of cell-based software architecture, specifically focusing on the number of cells and their internal organization as a community detection problem. In this context, a mathematical model for optimizing Cell-based software architecture was constructed. Additionally, given that this problem is NP-hard, we have developed a heuristic for solving large-scale problems.

The rest of the paper is organized as follows. Section 2 introduces various software architectures that can be applied in the software development process. Additionally, the Cell-based software architecture is presented, as well as the Community Detection problem and its application in different fields. Section 3 defines the problem and proposes a mathematical model for optimizing Cell-based software architecture, along with evalua-

tion and optimization results. The heuristic and numerical results are introduced in Section 4. Finally, discussion and conclusions are presented in Section 5.

2. Background

2.1. Software Architecture

When software design is concerned, various software architectures can be observed. For example, monolithic architecture represents a traditional software design approach. This architecture involves multiple modules that are executed together as a single unit at runtime, resulting in high coupling between the modules [10]. On the other hand, microservice architecture is an alternative to monolithic architecture [48]. In microservice architecture, each element is implemented as a separate microservice, which operates independently as a single unit at runtime. This approach results in low coupling between microservices [10], [48]. However, taking into account that each microservice is managed independently, microservice organization and communication must be carefully considered [24]. In addition, microservices typically require additional components for management, such as microservice orchestration and choreography [58], which can introduce additional complexity. Although monolith and microservice architecture can co-exist within the same system, researchers are exploring approaches decomposing and gradually transitioning from monolithic applications to microservices [2], [15], [42], [53]. Both monolithic and microservice architectures require infrastructure services (e.g., application server, database server, etc.), which can be either on-premises or cloud-based.

Another alternative to monolithic and microservice architectures is serverless software architecture, an approach that focuses on designing services related to specific business capabilities [56]. In this context, Functions-as-a-Service (FaaS) can be coded and deployed, while the underlying infrastructure is managed by the cloud provider [57]. Although this approach allows software engineers to focus on business functions, it results in a high degree of coupling with the infrastructure services provisioned by the cloud provider.

Based on the previous discussion, it can be stated that each software architecture has its own pros and cons that should be carefully considered during the software design process.

2.2. Cell-based Software Architecture

Cell-based architecture can be defined as a software architecture that incorporates multiple units of workload, with each unit known as a cell [55]. Each cell is independent from other cells, does not share state with other cells, and can encapsulate multiple components of different types [1], [55]. Additionally, each cell contains a cell gateway, serving as the central entry point for cell communication. In this context, intra-cell and inter-cell communication can be observed, which is realized with well-defined interfaces and protocols [1]. A specific set of functionalities or services can be incorporated within a cell, defining a cell boundary. In this context, cell-based architecture can be related with domain-driven software design [45].

Conceptual overview of the Cell-based software architecture is presented in Figure 1. The figure depicts two cells with multiple elements, with the cell boundaries outlined by

octagons. Cell A incorporates three elements (e.g., one monolith and two microservices), while Cell B also includes three elements (e.g., three microservices). Additionally, element A2 from Cell A communicates with Cell B through the cell gateway. In this way inter-cell communication is realized [1]. On the other hand, element B2 communicates with element B3. This communication is performed inside Cell B and represents intra-cell communication [1]. Each cell is autonomous and can be managed independently of other cells. As a result, better encapsulation, isolation, and distribution of software architecture elements can be achieved, addressing some of the typical challenges in software architecture design [9].

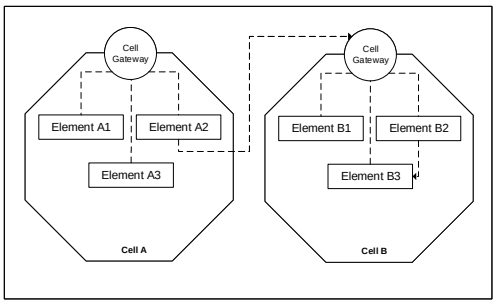

Fig. 1. Conceptual overview of the Cell-based software architecture

Considering that a cell can incorporate multiple capabilities implemented in various architectures, the cell-based approach facilitates the introduction of multi-architecture software development. In this context, the benefits of each applied architecture can be utilized, while their cons can be managed. This approach allows each cell to be independent and iterate individually, resulting in decentralized software architecture [1].

2.3. Community Detection Problem

Community detection problem belongs to the field of Complex Network Analysis. Its most common areas of application are: social networks [16], [38], neuroscience and biology [46], supply chain networks [41], [63], politics, customer segmentation, smart advertising and targeted marketing [30], etc. The community detection approach was also applied in software engineering, as both the process-oriented and object-oriented software architecture can be presented as a complex network [36] characterized by properties like those commonly observed in other complex networks [60]. Authors Pan, Jing, and Li used community detection approach for refactoring the package structures of object-oriented software in order to improve the maintenance process [49]. Software maintenance was also emphasized as the reason for using community detection in research conducted by Huang et al. [27]. Authors Hou, Yao, and Gong applied community detection approach to developer collaboration network in software ecosystem based on developer cooperation intensity [26].

In the context of software architecture, the Community Detection approach can be used to analyze service dependency graphs in microservice architectures [22]. This approach has been applied to facilitate the extraction of microservices from monolithic applications [20], [37]. Based on the defined entity types and their relationships, the Louvain algorithm for graph clustering was employed to identify potential microservice candidates [37]. The research conducted by Filippone et al. proposes microservices extraction by applying graph clustering and combinatorial optimization to maximize cohesion and minimize coupling between microservices [20].

Communities are groups of network's vertices with the common properties and/or role in the network [21]. The community detection problem is to find communities that maximize a given quality function. The solution of the problem is a set of communities such that the number of edges within the community is greater of the number of edges between the community's vertices and the rest of the network (Figure 2).

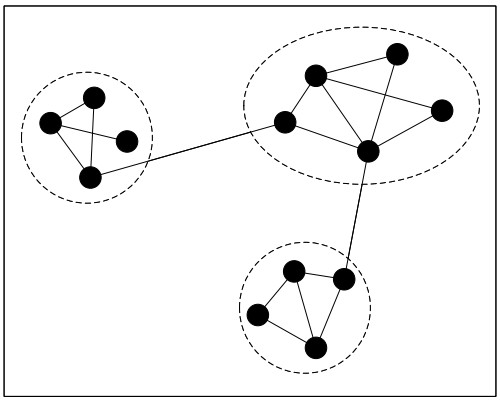

Fig. 2. A simple graph with three communities

There are several quality measures intended to evaluate the structure of communities [17]. In this paper we use the standard and most used measure of quality, called Newman–Girvan modularity [47]. One of the formulations of Newman–Girvan modularity is:

$$Q = \sum_{k \in C} [\frac{L_k}{L} - (\frac{D_k}{2L})^2] \tag{1}$$

where C represents the set of communities, L_k is the sum of the weights of the edges within community k, L is the sum of weights of all edges in the entire network, and D_k is the sum of the degree of the vertices in community k.

The function (1) is nonlinear and it can only be solved for small and medium-sized unweighted graphs [7]. Hence, several solving methods and linearizations of it can be found in the literature [54], [19]. In this paper, we use the variant of modularity function

(1) that enables further linearization, proposed by Alinezhad et al. in [5]:

$$Q = \frac{1}{L} \sum_{k \in C} (\sum_{(i,j) \in E_k} b_{ij} - \frac{1}{4L} \sum_{i,j \in V_k} d_i d_j) \tag{2}$$

where E_k represents the set of edges in the community k of a given graph $G = (E, V)$, V_k is the set of the vertices in the community k. The parameter b_{ij} is the weight of the edge (i, j), $(i, j) \in E$, and d_i is the weighted degree of the vetrex, obtained as sum of the weight of all input and output edges of vertex i:

$$d_i = \sum_{(i,j),(j,i) \in E} b_{ij}, i \in V \tag{3}$$

3. Mathematical Model for Optimizing Cell-based Software Architecture

As previously discussed in Section 2, cell-based software architecture can be depicted as a network of interconnected cells and elements that communicate with each other. Given that the solution of Community Detection problem can identify closely connected items, this section presents a mathematical model for optimizing cell-based software architecture. Optimizing the cell-based architecture can potentially lead to better resource utilization through an optimal number of cells and their internal organization. Additionally, various software quality attributes can be improved.

The model for optimizing Cell-based software architecture is related to the research presented in [44], whose core is based on the mathematical model presented in [5] by Alinezhad et al. However, in that study, we did not explain how the weights of the edges should be determined, as they were randomized in the experiments [44]. In this context, the model will be extended to define the measurement and calculation of these weights. Based on this extension, a software system will be developed, a simulation will be conducted, and the results will be analyzed. Given that this problem is NP-hard, a heuristic for solving large-scale problems will be developed, and its results will be compared with the optimization results obtained from the mathematical model.

When the software architecture whose elements should be grouped into cells based on community detection problem, the elements of the architecture are vertices of the graph $G = (E, V)$. The edges of the graph exist between the elements (vertices) which communicate, while the weight of the edge (i, j), b_{ij} represents the intensity of the communication.

To determine the intensity of communication, various software metrics can be utilized. These software metrics should be observed and collected within a specified time frame, allowing insight into application quality [51]. However, it is important to note that different software metrics may have different scales and measurement units [59], which must be carefully considered. For instance, response time is typically measured in time units (e.g. seconds, milliseconds, microseconds, nanoseconds, etc.), while payload size is typically measured in data units (e.g. bytes, kilobytes, megabytes, gigabytes, etc.). To ensure data consistency and maintain precision, it is essential to standardize measurement units

and perform necessary adjustments before further analysis [23], [50]. Based on the observed and collected metrics, we propose an approach that introduces normalization and aggregation of raw values. In this context, the Min-Max normalization can be applied:

$$m_{ij} = \frac{x_{ij} - min(X)}{max(X) - min(X)} \tag{4}$$

where x_{ij} represents the raw value of the examined software metric, X is the set of all raw values for that metric within the observed time frame, and m_{ij} is its normalized value within the range [0,1].

All relevant software metrics should contribute to the determination of the intensity of communication. However, some software metrics may have a greater impact than others. To account for this, a weighted sum approach can be utilized:

$$b_{ij} = w_1 m_{1ij} + w_2 m_{2ij} + ...w_n m_{nij} \tag{5}$$

where $m_{1ij}, m_{2ij}, ..., m_{nij}$ are normalized software metrics, $w_1, w_2, ..., w_n$ are weights that represent the importance of each metric, and b_{ij} represents the intensity of the communication.

Various software metrics can be utilized to determine the intensity of communication between software elements. In our approach, we will utilize software metrics related to effective interaction between software elements. Two software elements can interact through different communication models, protocols, and patterns [4]. For instance, in the Request-Response model, direct invocations can be calculated. On the other hand, in the Pub-Sub model, published messages received can be calculated. A high number of interactions between two software elements may indicate a high intensity of communication, as frequent exchanges suggest a strong relationship between the elements (in terms of interaction). If not properly managed, high communication can lead to performance anomalies and bottlenecks [28].

Furthermore, the time required for the submission of the request and the completion of the response should also be considered [51], [18]. Request Time refers to the duration taken to transmit the request from the sender software element to the receiver software element, including potential delays such as network latency and queueing delays. On the other hand, Response Time includes the actual time required for request processing, along with additional delays that may occur during transmission of the response (e.g., network delays, queueing delays). It is important to note that the request time and response time may differ based on the applied communication model (e.g. synchronous vs. asynchronous calls). High request time and response time may indicate network issues, inadequate resource utilization, or inefficient processing mechanisms [35], [64]. In this context, both request time and response time should be properly monitored to improve communication efficiency.

During each interaction, data is transmitted between software elements. For instance, a request may contain input data that needs processing, while a response may return output data as the result of the request. These data exchanges can be represented in various formats (e.g., plain text, JSON, XML, binary). Large transmitted data indicate substantial data transfer requirements [40]. If not properly managed, this can lead higher network utilization and increased bandwidth consumption. On the other hand, small transmitted

data suggests lightweight interactions, but has the potential to cause significant overheads [39].

The examined software metrics are commonly employed in contexts where communication between software systems is of primary concern. In addition, these metrics are supported by a variety of tools designed for the measurement and monitoring of software performance. Based on the discussed metrics, communication intensity proposed in equation (5) should be calculated as follows:

$$b_{ij} = w_1 m_{NoIntij} + w_2 m_{ReqTij} + w_3 m_{ResTij} + w_4 m_{ReqSij} + w_5 m_{ResSij} \tag{6}$$

where $m_{NoIntij}$ is normalized software metric Number of Interactions, m_{ReqTij} is normalized software metric Request Time, m_{ResTij} refers to normalized software metric Response Time, m_{ReqSij} represents normalized software metric Request Size, and m_{ResSij} is normalized software metric Response Size.

Considering different communication models, protocols, and patterns, as well as the importance of the examined software metrics, certain elements of equation (6) (i.e., weights or metric values) may vary in magnitude, including the possibility of being zero.

Since the weights inside the parentheses in equation (2) should be calculated only for the edges and vertices belonging to the same community, the set of communities C and binary variables y_{ik} are introduced:

$$y_{ik} = \begin{cases} 1, & \text{if vertice } i \text{ is in the community } k \\ 0, & \text{otherwise} \end{cases}$$

where $i \in V, k \in C$.

Equation (2) now becomes:

$$Q = \frac{1}{L} \sum_{k \in C} \Big(\sum_{(i,j) \in E_k} b_{ij} y_{ik} y_{jk} - \frac{1}{4L} \sum_{i,j \in V_k} d_i d_j y_{ik} y_{jk} \Big) \tag{7}$$

where L represents the sum of the weights of all edges of the entire network.

The nonlinearity $y_{ik} y_{jk}$ could be replaced by auxiliary binary variables z_{ijk}, where $k \in C, (i,j) \in E$:

$$z_{ijk} = \begin{cases} 1, & \text{if edge vertices } i,j \text{ are in the community } k \\ 0, & \text{otherwise} \end{cases}$$

and inequalities:

$$z_{ijk} \geq y_{ik} + y_{jk} - 1, k \in C, i,j \in V \tag{8}$$

$$z_{ijk} \leq y_{ik}, k \in C, i \in V, i,j \in V \tag{9}$$

$$z_{ijk} \leq y_{jk}, k \in C, j \in V, i,j \in V \tag{10}$$

The condition (8) ensures that variable z_{ijk} get the value 1 if both y_{ik} and y_{ik} have the value 1, i.e. the edge (i,j) is inside the community k if both vertices i and j belong to the community k. Since based on condition (8), value of z_{ijk} can be 1 if y_{ik} and/or y_{jk} are equal to zero, the conditions (9) and (10) are introduced to prevent such solutions. Furthermore, if for an edge (i,j) z_{ijk} equals zero for all $k \in C$, it indicates that edge

(i, j) does not belong to any community; instead, it represents a link between two different communities.

Additionally, the model also incorporates the concept of functionality. Functionality can be defined as a set of capabilities allowable and actionable by the software system [29]. Each functionality contains elements focused on a specific domain and should not be mixed to maintain boundaries, reduce complexity, and ensure modularity. In a cell-based software architecture, a single functionality can be represented by one or more cells, forming the foundation for optimizing the software architecture. In addition, different functionalities should not be organized in the same cell, allowing better separation of concerns between cells. As a result, each cell can be independent and managed individually [1].

In addition to the already introduced parameters and variables, notation used for the mathematical model formulation is as follows.

Sets

FC - set of functionalities

F_l - set of vertices of l-th functionality, $F_l \subset V, \bigcap_{l \in FC} F_l = \emptyset, \bigcup_{l \in FC} F_l = V$

Parameters

e – lower bound of the number of vertices in communities

Variables

$$x_k = \begin{cases} 1, & \text{if community } k \text{ exists} \\ 0, & \text{otherwise} \end{cases}$$

The final mathematical model, taken from [5] and extended with constraints related to functionalities, is listed below.

$$\max f(z) = \frac{1}{L} \sum_{k \in C} \Big(\sum_{(i,j) \in E_k} b_{ij} z_{ijk} - \frac{1}{4L} \sum_{i,j \in V_k} d_i d_j z_{ijk} \Big) \tag{11}$$

s.t.

$$z_{ijk} \geq y_{ik} + y_{jk}, k \in C, i, j \in V \tag{12}$$

$$z_{ijk} \leq y_{ik}, k \in C, i, j \in V \tag{13}$$

$$\sum_{k \in C} y_{ik} = 1, i \in V \tag{14}$$

$$y_{ik} \leq x_k, i \in V, k \in C \tag{15}$$

$$\sum_{i \in V} y_{ik} \geq e x_k, k \in C \tag{16}$$

$$y_{ik} + y_{jk} \leq 1, k \in C, i \in F_l, j \in F_p, l \neq p \tag{17}$$

$$x_k = \{0, 1\}, k \in C \tag{18}$$

$$y_{ik} = \{0, 1\}, k \in C \, i \in V \tag{19}$$

$$z_{ijk} = \{0, 1\}, k \in C \, i, j \in V \tag{20}$$

The objective function (11) represents the modularity measure linearized by replacing $y_{ik} y_{jk}$ with z_{ijk} in (7) . Since this function should be maximized, the first addend in parentheses will be as large as possible. Thus, the branches that have a greater weight

will be within the same community, that is, the software elements with more frequent communication will be in the same cell. Constraints (12) and (13) are related to linearization. Constraint (14) ensures that each vetrex is assigned exactly to one community. The constraint (15), the value 1 is set to x_k if some vertex is assigned to the k-th community. Constraint (16) is related to the minimal number of vertices assigned to the existing communities. Constraint (17) provides that vertices of different functionality cannot belong to the same community, i.e. only vertices of the same functionality can be in the same community. Constraints from (18) to (20) are related to the binary restrictions on the variables.

If necessary, additional constraints can be introduced. For example, although the parameter *e* defines the lower bound for the number of vertices in communities, an additional constraint can be added to specify a different minimum number of vertices for a particular community. This allows for fine-grained definition of cell structure in specific circumstances.

For example, if some of the software element should be isolated in a cell, a set of such element $VI \subset V$ and additional constraint can be included into mathematical model:

$$y_{ik} + y_{jk} \leq 1, k \in C, i \in VI, j \in V, i \neq j \tag{21}$$

Additionally, if some elements should be in the same cell, regardless the connections between them, the mathematical model can be extended as follows.
G - set of predefined groups of elements
V_q - set of elements predetermined to be in the same cell, $q \in G$

$$y_{ik} = y_{jk}, k \in C, i, j \in V_q, q \in G \tag{22}$$

$$y_{ik} + y_{jk} \leq 1, k \in C, i \in V_q, j \in V \setminus V_q \tag{23}$$

The constraint (22) ensures that the predetermined elements are in the same cell but allows other elements to be assigned to that cell as well. If it is necessary to assign to the same cell only the elements from $q \in G$, constraint (23) should be included into mathematical model.

3.1. Evaluation

For evaluation purposes, we have developed a software system related to a manufacturing domain (e.g., mobile phone manufacturing). The software architecture of this system is presented in Figure 3. Multiple software elements can be observed in the figure. The system incorporates five microservices (i.e., *inventory*, *product*, *order*, *notification*, and *payment*), as well as two monoliths (i.e., *legacy* and *production*). RESTful web services are used for communication between software elements, while the JSON format is used for data transfer. All software elements were executed within Docker containers, ensuring their portability across various operational environments [32]. Considering that containers are typically executed in isolation, we have created a virtual network for their communication (depicted as the *comsis-network* entity in the figure). Source code and installation instructions are provided in the Data Availability Statement section.

In order to determine the intensity of communication between these software elements, a simulation was conducted using Apache JMeter to create performance testing

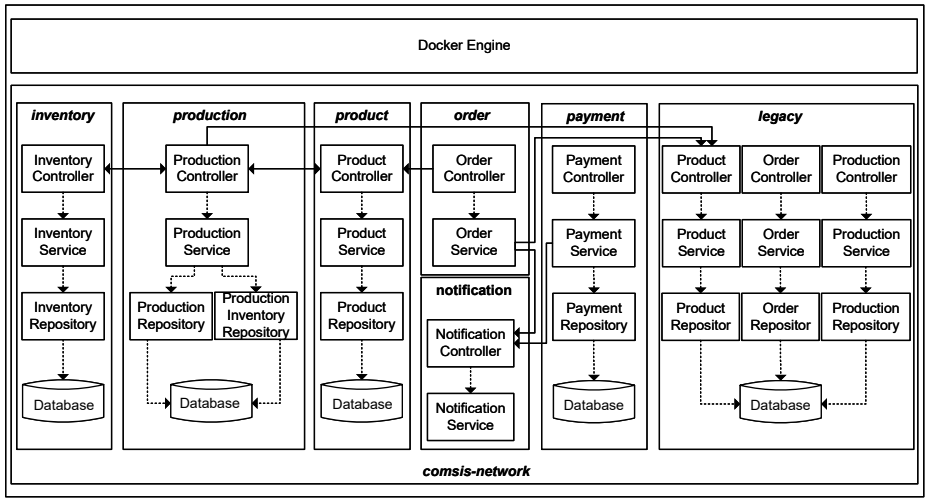

Fig. 3. Software architecture of the examined software system

plans [34]. The testing included a random number of invocations, threads, and ramp-up periods (i.e., delays) in communication, along with diverse input data sourced from CSV files containing mock data. For simulation purposes, all elements ran on a computer equipped with an Intel Core i7 (8th Generation) 2.00 GHz processor, 16 GB of RAM, and a 512 GB SSD drive. The simulation ran for 120 seconds, generating 336,858 effective interactions, which were then used to compute normalized software metrics. Table 7, located in the Appendix, presents the effective interactions between software elements, total and average values of software metrics, and their normalized counterparts. Based on the normalized metrics, the communication intensity – also shown in Table 7 – was calculated using equation (6), with each metric assigned an equal weight. The simulation test plan and results are available in the Data Availability section. The optimized structure of the solution can be graphically presented based on the results. Figure 4 presents the results of the optimization of a manufacturing software system (e.g., mobile phone manufacturing). The input includes defined Production and Purchasing functionalities. The Production functionality comprises two monolithic applications (i.e., $production$ and $legacy$) and two microservices (i.e., $inventory$ and $product$), while the Purchasing functionality consists of three microservices (i.e., $order$, $payment$, and $notification$). Additionally, the communication between these elements is specified (the weights of the edges in Figure 4).

Based on the performed optimization, the resulted solution includes three communities (named Community A, Community B, and Community C), each containing different elements (see Figure 4). In the following text these communities will be referred to as the Production Cell (Community A), Purchasing Cell (Community B), and Legacy Cell (Community C).

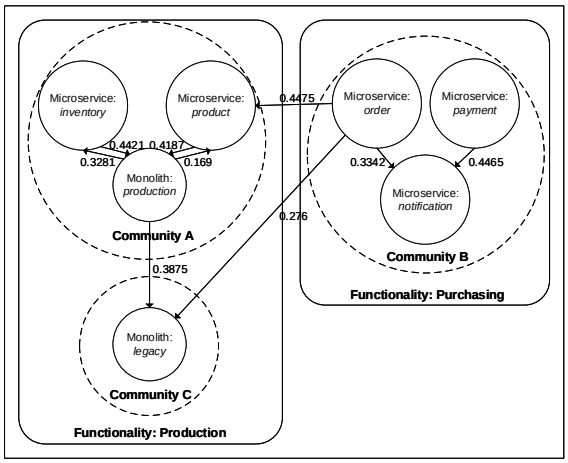

Fig. 4. Results of the optimization of a manufacturing software system

An additional observation pertains to the Legacy Community, which contains only one element (i.e., the $legacy$ monolith). A legacy software system is defined as a core system that has been functioning correctly in production for decades [33]. Considering the prevalence of legacy systems today, researchers are exploring approaches to migrate these systems to modern architectures [3], [31], [43]. In the context of cell-based software architecture, the legacy element is incorporated within a specific cell. From the optimization model perspective, this is represented as an additional constraint that restricts the particular cell structure:

$$y_{legk} + y_{jk} \leq 1, k \in C, j \in V \setminus leg \tag{24}$$

where leg is the index of the variable corresponding to the legacy monolith. This constraint ensures that no other element can be in the cell containing the legacy monolith.

Another interesting observation pertains to the Cell Gateway component. While this component is not explicitly represented in the mathematical model, it serves as the central entry point for cell communication [1]. Therefore, we have included one gateway per cell based on the optimal solution. The final software architecture is shown in Figure 5.

In order to validate the mathematical model, we conducted a series of experiments. The evaluation considered three functionalities, with the number of elements varied. Each element represents an instance of software architecture encapsulating specific features (e.g., a monolith with multiple features, a microservice containing a single feature). Considering that these elements can vary in terms of their applied software architecture and size, a software system can encompass numerous elements. Within this context, the first functionality incorporated 40 percents of the elements, while the remaining elements were equally divided between the other two functionalities. Taking into account that elements cooperate with each other, each element has at least one connection with another element within the same functionality, while up to 40 percents of elements have double connections within the same functionality. Finally, considering that all functionalities are part of

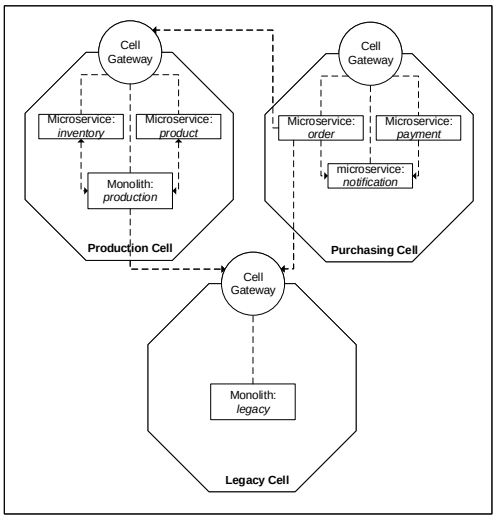

Fig. 5. Cell-based software architecture based on the optimal solution

the same software system, two connections between elements from different functionalities are also established. The elements previously discussed, such as the edges and their weights (the intensity of communication between the elements - parameter b_{ij}) are obtained using equation (6), where the metrics values are randomly generated as explained for the manufacturing software system. The lower bound of the number of vertices in communities parameter e is set to 2.

All optimizations were performed solved using GLPK software on a laptop computer equipped with 11th Gen Intel(R) Core(TM) i5 and 16 GB of RAM. The solving method used in the GLPK software was Branch and Cut, with Gomory's mixed integer cuts, MIR (mixed integer rounding) cuts, mixed cover cuts, and clique cuts options.

The first experiment was conducted to determine whether the metric weights in equation (6) affect the grouping of the elements into cells. The optimization was performed for a graph with 30 vertices and 38 edges in eight scenarios. In the first scenario, the weight of all metrics is equal (0.2). In scenarios two through six, only one metric is considered important to the decision maker. In the seventh scenario only the request and response time metrics are important (with w_2 and w_3 equal to 0.5). In the last scenario, only the request and response size metrics are important (with w_4 and w_5 equal to 0.5). The results of the optimizations are given in Table 1, where the rows represent eight scenarios, and the columns refer to three functionalities and the grouping of its elements.

Table 2 shows the results of the grouping when certain metrics are not important to the decision maker. In the first five cases, the weights w_1 to w_5 from the equation (6) are equal to zero, that is, corresponding metrics are considered irrelevant. In the sixth scenario, the request and response time are not important (with w_2 and w_3 equal to zero), while the last scenario refers to the irrelevance of the request and response size (with w_4 and w_5 equal to zero).

Table 1. Grouping into communities depending on metric weights (important metrics)

Case	f1	f2	f3
equal	{1,2,4,5,7,8,11}{3,6,9,10,12}	{13,14,15,16,17,19,20}{18,21}	{22,23,24,26,28,29,30}{25,27}
w_1	{1,5}{2,4,7,8,11}{3,6,9,10,12}	{13,16,17,20}{14,15,19}{18,21}	{22,23,26,29}{24,28,30}{25,27}
w_2	{1,2,4,5,7,8,11}{3,6,9,10,12}	{13,14,15,16,17,19,20}{18,21}	{22,23,29,30}{24,26,28}{5,27}
w_3	{1,2,4,5,7,8,11}{3,6,9,10,12}	{13,16,17,19,20}{14,15}{18,21}	{22,23,24,26,28,29,30}{25,27}
w_4	{1,5}{2,4,7,8,11}{3,6,9,10,12}	{13,14,15,16,17,19,20}{18,21}	{22,23,24,26,28,29,30}{25,27}
w_5	{1,5}{2}{3,6,9,10,12}{4,7,8,11}	{13,20}{14,15,16,17,19}{18,21}	{22,23,24,26,28,29,30}{25,27}
w_2w_3	{1,2,4,5,7,8,11}{3,6,9,10,12}	{13,14,15,16,17,19,20}{18,21}	{22,23,24,26,28,29,30}{25,27}
w_4w_5	{1,2,4,5,7,8,11}{3,6,9,10,12}	{13,14,15,16,17,19,20}{18,21}	{22,23,24,26,28,29,30}{25,27}

Table 2. Grouping into communities depending on metric weights (irrelevant metrics)

Case	f1	f2	f3
w_1	{1,2,3,4,6,8,10,11,12}{5}{7}{9}	{13,15,16,17,18,21}{14,19,20}	{22,23,24,26,28}{25,27}{29,30}
w_2	{1,2,4,8,11}{3,6,10,12}{5,9}{7}	{13,14,15,16,17}{18,21}{19,20}	{22,23,24,26,28}{25,27}{29,30}
w_3	{1,2,4,7,8,11}{3,5,6,10,12}{9}	{13,15,16,17}{14,18,19,20,21}	{22,23,24,26,28}{25,27}{29}{30}
w_4	{1,2,3,4,5,6,8,10,11,12}{7}{9}	{13,15,16,17}{14,19,20}{18,21}	{22,23,24,26,28,29}{25,27}{30}
w_5	{1,2,4,7,8,11}{3,6,10,12}{5}{9}	{13,14,15,16,17,19,20}{18,21}	{22,23,24,26,28}{25,27}{29}{30}
w_2w_3	{1,2,3,4,6,8,10,11,12}{5}{7}{9}	{13,14,15,16,17,18,19,20,21}	{22,23,24,25,26,27,28}{29}{30}
w_4w_5	{1,2,4,5,9,11}{3,6,10,12}{7,8}	{13,14,15,16,17}{18,21}{19,20}	{22,23,24,25,26,27,28}{29}{30}

Based on the clustering in Tables 1 and 2, it can be concluded that the metric weights influence the resulting clustering. This effect is especially evident when comparing the grouping results in the case of extreme importance of a given metric ($w_i = 1, i = \overline{1,5}$) and its complete irrelevance ($w_i = 0, i = \overline{1,5}$). Although there are the same groupings within individual functionalities, they differ at the level of the entire software. The only exception occurs when multiple metrics are equally important (cases "equal", w_2w_3, and w_4w_5 in Table 1).

Since no optimal solutions were obtained for the 5-minute time limit in any case, the quality of the obtained solutions was analyzed based on the gap relative to the LP relaxation of the problem. The LP relaxation of the mathematical model (11)-(20) was obtained by omitting the integer constraints (18)-(20). Figure 6 shows the modularity values (the value of the objective function (11)) for the solution obtained after 5 minutes of optimization, as well as for the LP relaxation solution. The first eight cases correspond to those in Table 1, while the remaining seven refer to the cases in Table 2.

The average gap value is 4.8%. The smallest value of the gap of 0.7% is obtained for the case when the weights w_4 and w_5 are equal to 0.5, while the smallest refers to the case when w_1 is equal to 1. It can be concluded that the obtained solutions are in most cases close to, or in some cases equal to the optimal solution. The 5-minute time limit was sufficient to obtain high-quality solutions, but not enough to finish the search of the entire solution space.

To examine the dimensions of the problem that can be solved through optimization and the robustness of the mathematical model, the next experiments were conducted for the graphs whose dimensions are given in Table 3. The columns named vertices and edges give the number of vertices and edges, respectively. In column L, the range of values of the parameter L in 10 instances is shown.

Given that the model parameters are randomly generated, the mathematical model (11)-(20) was applied to ten instances of each graph in Table 3. The execution time was

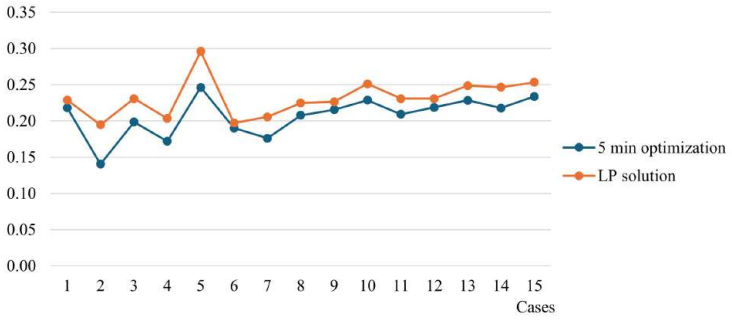

Fig. 6. The modularity obtained after 5 minutes optimization and the LP solution

Table 3. Summary of data sets used in the experiments

Case	Number of vertices	Number of edges	L
1	25	37	18.09-20.47
2	30	38	17.88-20.10
3	35	55	26.77-30.18
4	40	52	24.24-28.44
5	45	71	34.02-37.58
6	50	64	30.09-35.29
7	60	77	35.58-40.51
8	70	90	43.73-48.38
9	80	103	47.55-54.21
10	90	116	56.74-61.05
11	100	128	60.73-67.70
12	150	193	94.76-99.77
13	200	257	122.12-132.84
14	300	498	241.21-249.47

limited to 5 minutes. The average results are shown in Table 4, where the column Dim represents the number of vertices. The columns Number_of_cells and Max_cell_size represent the average number of cells and the average maximal size of the cell within 10 instances. The column Modularity shows the average modularity. The last three columns are related to the gap relative to the solution of the LP relaxation: the average gap, standard deviation and 95% confidence interval.

For the given 5-minute optimization limit time, it was possible to obtain a feasible (integer) solution only for 25-50 vertices. Figure 7 shows the value of the gap in all the instances of these six cases.

The largest gap was obtained in the second instance of the 45-vertices case, while the smallest gap was obtained in the eighth instance of the 50-vertices case. However, regardless of relatively large gaps, the mathematical model showed robustness when looking at the grouping of gaps within the same case (Figure 7).

Since an integer solution could not be obtained in five minutes for cases of 60 or more vertices, the next step is to develop heuristics to solve larger-dimensional problems.

Table 4. Average results of the optimization with limited time

Dim	Number of cells	Max cell size	Modularity	Gap		
				average(%)	SD	CI(%)
25	3.40	5.9	0.6596	26.37	0.0101	[25.65,27.10]
30	6.10	7	0.7932	13.2	0.0063	[12.73,13.64]
35	6.00	9.7	0.6368	30.9	0.0270	[28.93,32.79]
40	7.3	7	0.8377	10.5	0.0091	[9.85,11.15]
45	8.3	10.9	0.6041	35.7	0.0542	[31.78,39.54]
50	10.7	7	0.8359	11.9	0.0395	[9.07,14.73]

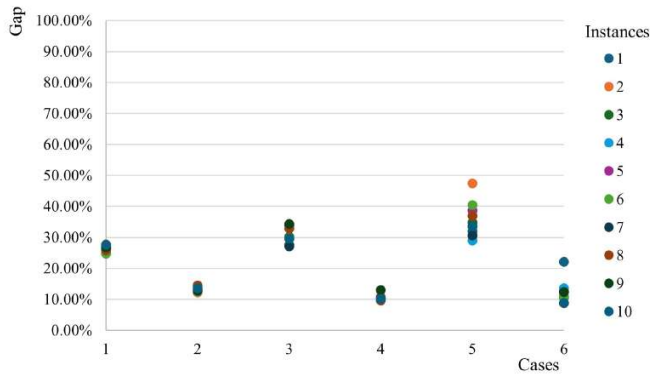

Fig. 7. The gap between the solution obtained after 5 minutes and the LP solution

4. Heuristic Approach to Problem Solving

As discussed in Section 3, functionalities serve as the foundation for optimizing the cell-based software architecture. Given that distinct functionalities should not be grouped within the same cell and that each functionality can span multiple cells, the following solution approach focuses on optimizing each functionality individually. By addressing each functionality separately, complexity is reduced while preserving clear boundaries and enforcing the separation of concerns between functionalities and cells [1]. Consequently, the following algorithms should be applied iteratively for each functionality, with the results (i.e., modularities and identified communities) analyzed collectively to derive the optimal architecture for the entire software system.

First, we will explain the auxiliary *Greedy algorithm*. The main idea of our *Greedy algorithm* is shown in Figure 8. Using the *Greedy algorithm*, we increase the current modularity by merging (uniting) selected communities. Let C' be a set of communities, and $k, l \in C', k \neq l$. The new set of communities obtained by merging the communities k and l, while leaving the other communities unchanged is denoted as C_{kl}. Let Q_{kl} represent the modularity of C_{kl}.

Below is the pseudocode of our *Greedy algorithm*. First, for all pairs $k, l \in C'$ we determine C_{kl} with modularity Q_{kl} (line 2-3) and determine the largest among them Q_{rs} (line 4). If the obtained value is greater than the current modularity Q then the current community C with modularity Q becomes C_{rs} with modularity Q_{rs} (line 5-6). The de-

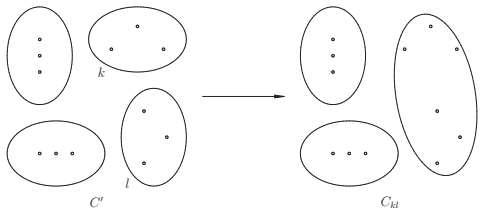

Fig. 8. Cell merging process in the *Greedy algorithm*

scribed procedure is repeated as long as it is possible to improve the modularity Q (loop 1-7).

$Greedy\ algorithm$

Input: Graph G and set of communities C with modularity Q.
Output: New set of communities C' with modularity Q'.
Initialization $C' = C$ and $Q' = Q$;
1: **repeat**
2: **for all** $k, l \in C', k \neq l$ **do**
3: Determine C_{kl} with modularity Q_{kl};
4: Determine $Q_{rs} = \max_{k,l \in C, k \neq l} Q_{kl}$;
5: **if** $Q_{rs} > Q'$ **then**
6: $Q' = Q_{rs}$ and $C' = C_{rs}$;
7: **until** $Q' \neq Q_{rs}$;
Print out C' with modularity Q'.

Now, with the *Destroy and repair algorithm*, we increase current modularity by alternately disassembling and merging cells. The pseudocode of the algorithm is given below. The initial set of communities C^*, with modularity Q^*, is obtained by applying the *Greedy algorithm* on the trivial set of communities $C = \{\{i\} : i \in V\}$ (line 1). We randomly select two communities k and l (line 3), which we then decompose into individual vertices, while the other cells remain unchanged (Figure 9). Let $\overline{C_{kl}}$ denote the set of communities obtained in this manner, with modularity $\overline{Q_{kl}}$ (line 4.).

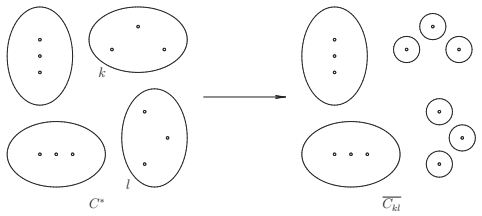

Fig. 9. Cell decomposition process in the *Destroy and repair algorithm*

By applying the *Greedy algorithm*, the communities are reconnected, resulting in a new set of communities C' with modularity Q' (line 5). If the new modularity Q' exceeds the current best modularity Q^*, then the destroying and repair process was successful and the set of communities C', with modularity Q', becomes the new best (line 6-7). Otherwise, the counter of unsuccessful attempts to destroy and repair is increased (line 8-9). Since the process of destroying is stochastic (line 3), the described process of destroying and repairing is repeated until the maximum number of consecutive unsuccessful attempts $Count_{max}$ is reached (loop 2-10).

Destroy and repair algorithm

Input: Graph G, $Count_{max}$.
Output: Set of communities C^* with modularity Q^*.
Initialization: $C = \{\{i\} : i \in V\}$ and $Q = Counter = 0$.
1: $(C^*, Q^*) = Greedy(G, C, Q)$;
2: **repeat**
3: Randomly choose $k, l \in C^*, k \neq l$;
4: Determine $\overline{C_{kl}}$ with modularity $\overline{Q_{kl}}$;
5: $(C', Q') =$ ***Greedy*** $(G, \overline{C_{kl}}, \overline{Q_{kl}})$;
6: **if** $Q' > Q^*$ **then**
7: $(C^*, Q^*) = (C', Q')$ and $Counter = 0$;
8: **else**
9: $Counter = Counter + 1$;
10: **until** $Counter = Count_{max}$;
Print out C^* with modularity Q^*.

So, with the *Destroy and repair algorithm*, we alternately disassemble and assemble the community structure in order to obtain a structure with as much modularity as possible.

4.1. Evaluation of the heuristic

The heuristic algorithm was evaluated on the graphs given in Table 3. The average results are shown in Tables 5 and 6. Table 5 contains the same elements as Table 4: the number of vertices, the average number of cells, the average maximal cell size, the average modularity, the average gap, standard deviation, and 95% confidence interval. Data on heuristic execution times are given in Table 6: average time in seconds, standard deviations, and 95% confidence interval.

The detailed data for all instances of 14 cases are given in Figures 10 and 11. Figure 10 shows the gap values, while Figure 11 shows the time to reach the algorithm's stopping criterion.

Based on Figures 10 and 11, it can be concluded that the heuristic is robust. Changes in parameters do not affect the quality of the solution or the time required to obtain it when the heuristic is applied to graphs of the same dimensions.

Finally, heuristic was used to examine the impact of metric weighting schemes on community structures. For that purpose, we have generated a new graph consisting of

Table 5. Average results of the heuristic

Dim	Number of cells	Max cell size	Modularity	Gap		
				average(%)	SD	CI(%)
25	3.40	9.4	0.6596	26.37	0.0101	[25.65,27.1]
30	6.10	7	0.7932	13.18	0.0063	[12.73,13.64]
35	4.10	12.1	0.6730	26.93	0.0112	[26.13,27.74]
40	7.1	7	0.8399	10.26	0.0032	[10.04,10.49]
45	4.8	13.8	0.6840	27.13	0.0139	[26.15,28.13]
50	9	7	0.8311	12.40	0.0024	[12.24,12.58]
60	10	7	0.8916	6.84	0.0016	[6.73,6.96]
70	10	7	0.8988	6.66	0.0007	[6.61,6.72]
80	13.0	7	0.9179	5.19	0.0011	[5.11,5.27]
90	14.0	7	0.9148	5.84	0.0018	[5.72,5.98]
100	16.0	7	0.9313	4.39	0.0006	[4.35,4.44]
150	23.0	7	0.9545	2.89	0.0003	[2.87,2.92]
200	30.0	7	0.9656	2.18	0.0002	[2.17,2.2]
300	16	20	0.9354	5.56	0.0003	[5.55,5.59]

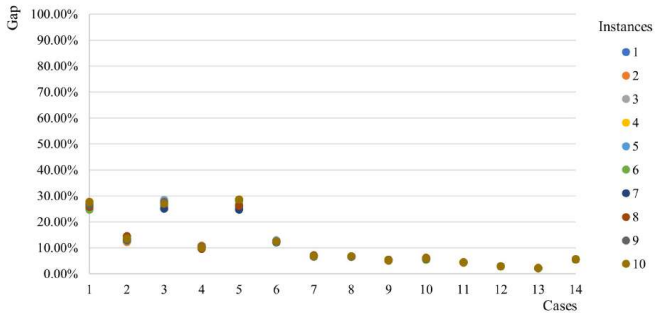

Fig. 10. The gap between the heuristic solution and the LP solution

30 vertices and 38 edges. Using the *Destroy and repair algorithm*, we have solved 15 scenarios. Table 8, incorporated in the Appendix, shows first eight scenarios: 1. the weight of all metrics is equal (0.2), in scenarios 2-6. only one of metric w_1-w_5 respectively have the weight equal to 1, while the other metrics have the weight 0, in scenario 7 the weights of w_2 and w_3 are equal to 0.5, and in scenario 8 the weights of w_4 and w_5 are equal to 0.5. Table 9, located in the Appendix, shows the last seven scenarios: in the first five columns only one metric is not important and have the weight 0, and in the last two columns, the weighst w_2,w_3, and w_4,w_5 are equal to 0, respectively. In all seven cases the weights of the remaining metrics are equal.

The community structures for given scenarios as well their modularity are given in sub-figures in Figure 12 and their captions. The missing scenarios have the same community structure as scenario w equal. However, their modularity differs from the modularity of the w equal scenario due to the different weights of the edges. The modularity of the missing scenarios: $w_5 = 1$, $w_4 = w_5 = 0.5$, $w_1 = 0$, $w_2 = 0$, $w_3 = 0$, $w_5 = 0$, and $w_2 = w_3 = 0$ are: 0.717, 0.69, 0.663, 0.668, 0.671, 0.648, and 0.68, respectively.

Table 6. Heuristic execution times

Dim	average(sec)	SD	CI(sec)
25	0.21	0.0170	[0.21,0.23]
30	0.22	0.0095	[0.22,0.23]
35	0.23	0.0261	[0.21,0.25]
40	0.24	0.0079	[0.23,0.25]
45	0.32	0.0301	[0.3,0.35]
50	0.31	0.0242	[0.3,0.33]
60	0.35	0.0262	[0.33,0.37]
70	0.43	0.0279	[0.42,0.46]
80	0.53	0.0581	[0.49,0.58]
90	0.79	0.0817	[0.73,0.85]
100	0.88	0.1056	[0.81,0.96]
150	2.48	0.1658	[2.36,2.6]
200	6.42	0.2716	[6.23,6.62]
300	59.88	3.8869	[57.11,62.67]

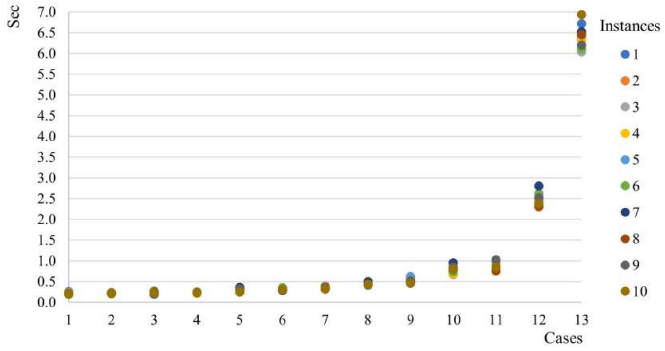

Fig. 11. Heuristic solution finding times

It can be concluded that different metric weights influence the detected community structures, as well as the values of modularity, even when the same community structure is obtained. In the examined case, however, these differences are relatively small. The standard deviation of modularity across all 15 scenarios is 0.022 (corresponding to 3.34% of the average value), and the 95% confidence interval is [0.653, 0.677]. These modest variations can be partly explained by the fact that the differences between the metric values—and thus the resulting edge weights—were not substantial. In cases where the edge weights differ more strongly, the variations in modularity may become more pronounced and potentially more relevant for decision-making.

5. Discussion and Conclusions

In this paper, the problem of grouping software elements into cells is addressed as a community detection problem – an optimization problem whose goal is to find a set of

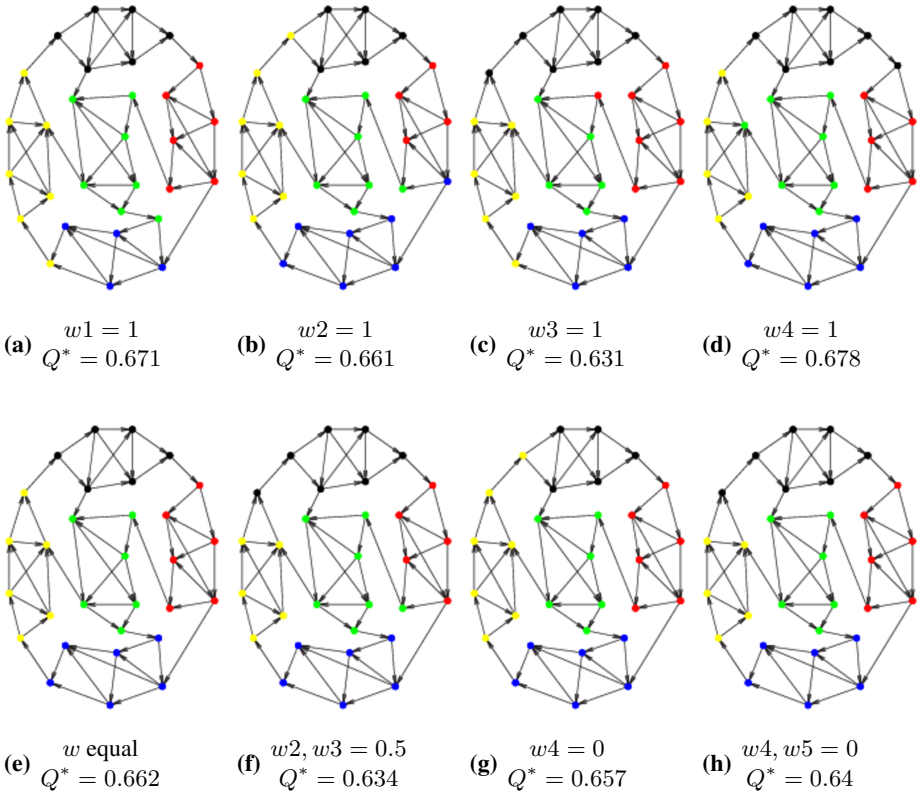

Fig. 12. The community structures for given scenarios

communities such that the connections between nodes within the community are stronger than the connections with other nodes in the graph. The elements of software are represented by vertices, communication between them by edges, and cells by communities. Following this analogy, a corresponding mathematical model was formulated.

Our model leverages a diverse set of software metrics and their normalization to accurately determine the intensity of communication, represented by the b_{ij} parameter. By employing a weighted sum approach, the model allows software architects to adjust metric priorities based on the specific characteristics of the software system being designed. For instance, in time-critical systems, higher weights can be assigned to Request Time and Response Time, ensuring that performance-related factors take precedence. Conversely, for data-intensive applications, greater emphasis can be placed on Request Size and Response Size to reflect the importance of data exchange. Additionally, the model is flexible, enabling the incorporation of additional software metrics and the customization of their weights. This adaptability influences the granularity and composition of cells, allowing for dynamic adjustments that align with evolving system requirements. As a continuous process, this approach promotes optimal resource utilization while maintaining architectural efficiency. The mathematical model formulated in this paper, in addition to determining the clusters, enables experimentation with the importance of individual metrics.

The objective of the mathematical model is the modularity function, which is originally nonlinear. Therefore, the linearization of this function was applied to ensure the possibility of obtaining solutions to problems of larger dimensions. However, despite linearization, integer (non-optimal) solutions were only obtained for graphs with up to 50 nodes within the 5-minute time limit. That is why heuristics were developed which have proven effective in solving the problem of grouping software elements into cells. Figure 13 shows a comparison of the modularity values obtained through 5-minute limited optimization and the heuristic, while Figure 14 gives the gaps for these two groups of the results.

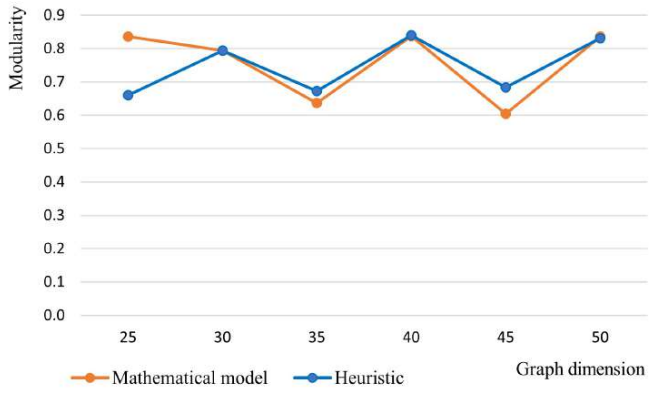

Fig. 13. Modularity function of 5-munites optimization and heuristc

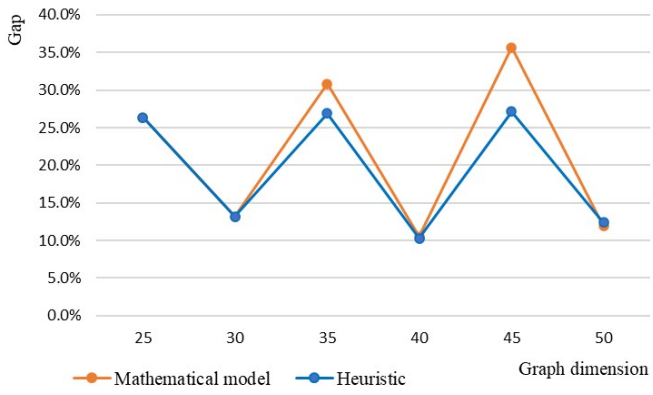

Fig. 14. Gaps in 5-munites optimization and heuristic

The modularity obtained by the heuristic is the same or better than the modularity obtained by the optimization limited to 5 minutes. Except for dimensions 25 and 30, where optimization and heuristics give the same results, the gap with respect to the LP model solution is much smaller for heuristics than for 5-minute optimization for all other dimensions. It is important to emphasize that the time required to obtain a solution using heuristics is, in all cases, less than one second, which is significantly shorter than the optimization time. Of course, it is possible to use a different (larger) counter value. It is also possible to select more than two communities in the destroying process. These changes would improve the quality of the obtained solutions, while increasing the time, that would still remain shorter than the optimization time.

The main goal of this research was to investigate the validity of modeling the cell-based software architecture as a community detection problem. The research has certain limitations, which open up space for further improvements. It is important to acknowledge the limitations of using the normalization. Min–max normalization is sensitive to extreme values, which can stretch the scale and compress most data into a narrower interval, potentially reducing variability that may be important for detecting subtle differences in edge strength. Furthermore, the final result depends on the choice of weights; If the weights are not well justified empirically or theoretically, the detected communities may reflect the assumptions of the researcher rather than the intrinsic structure of the network. In short, normalization and weighting together enhance methodological transparency and balance, but interpretation of the resulting communities should always take into account the assumptions underlying these preprocessing steps. In addition, destroying only two communities in the *Destroy and repair algorithm* gave good results in a very short time for graphs up to 200 nodes. However, for a graph of 300 nodes, the execution time increased by an order of magnitude (from 6.42 to 59.88 seconds). Subsequently, problems with 500 and 1000 nodes were solved to examine how the execution time further increases. The execution time of the heuristic for the graph of 500 nodes was 169.98 seconds, while for the graph with 1000 nodes it was about 40 minutes, which may indicate a computational problem for larger graphs.

Further research could examine system workload in the context of additional non-functional attributes such as scalability, availability, and reliability. Additionally, different linearization of the modularity function can be examined as well as different quality measures. In order to speed up the heuristic, the possibility of using other neighborhood structures than destroying only two communities should be examined. In addition, the efficiency of some other heuristics in solving the observed problem will be tested, primarily Variable neighborhood search.

References

1. Abeysinghe, A.: Cell-based architecture: A decentralized reference architecture for cloud-native applications. Available online: https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md (2024), accessed: 2024-05-17
2. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., Clarke, P.: Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering 49(8), 4213–4242 (2023)

3. Ahmad, A., Babar, M.A.: A framework for architecture-driven migration of legacy systems to cloud-enabled software. In: Proceedings of the WICSA 2014 Companion Volume, pp. 1–8 (2014)
4. Aksakalli, I.K., Çelik, T., Can, A.B., Tekinerdoğan, B.: Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software 180, 111014 (2021)
5. Alinezhad, E., Teimourpour, B., Sepehri, M.M., Kargari, M.: Community detection in attributed networks considering both structural and attribute similarities: two mathematical programming approaches. Neural Computing and Applications 32, 3203–3220 (2020)
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional, 4th edn. (2021)
7. Bennett, L., Liu, S., Papageorgiou, L.G., Tsoka, S.: A mathematical programming approach to community structure detection in complex networks. In: Computer Aided Chemical Engineering, vol. 30, pp. 1387–1391. Elsevier (2012)
8. Besker, T., Martini, A., Bosch, J.: Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software 135, 1–16 (2018)
9. Bierska, A., Buhnova, B., Bangui, H.: An integrated checklist for architecture design of critical software systems. In: FedCSIS (Position Papers). pp. 133–140 (2022)
10. Blinowski, G., Ojdowska, A., Przybyłek, A.: Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE access 10, 20357–20374 (2022)
11. Bollinadi, M., Damera, V.K.: Cloud computing: security issues and research challenges. Journal of Network Communications and Emerging Technologies (JNCET) 7(11) (2017)
12. Bourque, P., Fairley, R.E.E.: Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press (2014)
13. Brenner, R.: Balancing resources and load: Eleven nontechnical phenomena that contribute to formation or persistence of technical debt. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 38–47. IEEE (2019)
14. Chandrasekar, A., Rajesh, S., Rajesh, P.: A research study on software quality attributes. International Journal of Scientific and Research Publications 4(1), 14–19 (2014)
15. Chen, R., Li, S., Li, Z.: From monolith to microservices: A dataflow-driven approach. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). pp. 466–475. IEEE (2017)
16. Costa, A.R., Ralha, C.G.: Ac2cd: An actor–critic architecture for community detection in dynamic social networks. Knowledge-Based Systems 261, 110202 (2023)
17. Dao, V.L., Bothorel, C., Lenca, P.: Community structure: A comparative evaluation of community detection methods. Network Science 8(1), 1–41 (2020)
18. Eisty, N.U., Thiruvathukal, G.K., Carver, J.C.: A survey of software metric use in research software development. In: 2018 IEEE 14th international conference on e-Science (e-Science). pp. 212–222. IEEE (2018)
19. Ferdowsi, A., Chenary, M.D.: Toward an optimal solution to the network partitioning problem. In: 2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS). pp. 111–117. IEEE (2023)
20. Filippone, G., Mehmood, N.Q., Autili, M., Rossi, F., Tivoli, M.: From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In: 2023 IEEE 20th International Conference on Software Architecture (ICSA). pp. 47–57. IEEE (2023)
21. Fortunato, S.: Community detection in graphs. Physics reports 486(3-5), 75–174 (2010)
22. Gaidels, E., Kirikova, M.: Service dependency graph analysis in microservice architecture. In: Perspectives in Business Informatics Research: 19th International Conference on Business Informatics Research, BIR 2020, Vienna, Austria, September 21–23, 2020, Proceedings 19. pp. 128–139. Springer (2020)

23. Gudivada, V., Apon, A., Ding, J.: Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations. International Journal on Advances in Software 10(1), 1–20 (2017)
24. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: A self-adaptive roadmap. In: 2016 IEEE International Conference on Services Computing (SCC). pp. 813–818. IEEE (2016)
25. Hasselbring, W.: Software architecture: Past, present, future. The essence of software engineering pp. 169–184 (2018)
26. Hou, T., Yao, X., Gong, D.: Community detection in software ecosystem by comprehensively evaluating developer cooperation intensity. Information and Software Technology 130, 106451 (2021)
27. Huang, G., Zhang, P., Zhang, B., Yin, T., Ren, J.: The optimal community detection of software based on complex networks. International Journal of Modern Physics C 27(08), 1650085 (2016)
28. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Computing Surveys (CSUR) 48(1), 1–35 (2015)
29. ISO/IEC/IEEE: 24765:2017 systems and software engineering — vocabulary. Available online: https://www.iso.org (2017), accessed: 2024-05-27
30. Karataş, A., Şahin, S.: Application areas of community detection: A review. In: 2018 International congress on big data, deep learning and fighting cyber terrorism (IBIGDELFT). pp. 65–70. IEEE (2018)
31. Kazanavičius, J., Mažeika, D.: Migrating legacy software to microservices architecture. In: 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream). pp. 1–5. IEEE (2019)
32. Keller Tesser, R., Borin, E.: Containers in hpc: a survey. The Journal of Supercomputing 79(5), 5759–5827 (2023)
33. Khadka, R., Batlajery, B.V., Saeidi, A.M., Jansen, S., Hage, J.: How do professionals perceive legacy systems and software modernization? In: Proceedings of the 36th International Conference on Software Engineering. pp. 36–47 (2014)
34. Kiran, S., Mohapatra, A., Swamy, R.: Experiences in performance testing of web applications with unified authentication platform using jmeter. In: 2015 International Symposium on Technology Management and Emerging Technologies (ISTMET). pp. 74–78. IEEE (aug 2015)
35. Kraft, S., Pacheco-Sanchez, S., Casale, G., Dawson, S.: Estimating service resource consumption from response time measurements. In: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools. pp. 1–10 (2009)
36. Li, D., Han, Y., Hu, J.: Complex network thinking in software engineering. In: 2008 International Conference on Computer Science and Software Engineering. vol. 1, pp. 264–268. IEEE (2008)
37. Li, Z., Shang, C., Wu, J., Li, Y.: Microservice extraction based on knowledge graph from monolithic applications. Information and Software Technology 150, 106992 (2022)
38. Lin, C.C., Kang, J.R., Chen, J.Y.: An integer programming approach and visual analysis for detecting hierarchical community structures in social networks. Information Sciences 299, 296–311 (2015)
39. Lin, S., Yu, J., Jiang, X.: Signalling overhead analysis of small data transmission for machine type communication. In: 2018 IEEE 18th International Conference on Communication Technology (ICCT). pp. 664–668. IEEE (2018)
40. Liu, S., Jung, E.S., Kettimuthu, R., Sun, X.H., Papka, M.: Towards optimizing large-scale data transfers with end-to-end integrity verification. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 3002–3007. IEEE (2016)
41. Lu, Z., Dong, Z.: A gravitation-based hierarchical community detection algorithm for structuring supply chain network. International Journal of Computational Intelligence Systems 16(1), 110 (2023)

42. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS). pp. 524–531. IEEE (2017)
43. Menychtas, A., Santzaridou, C., Kousiouris, G., Varvarigou, T., Orue-Echevarria, L., Alonso, J., Gorronogoitia, J., Bruneliere, H., Strauss, O., Senkova, T., Pellens, B., Stuer, P.: Artist methodology and framework: A novel approach for the migration of legacy software on the cloud. In: 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 424–431. IEEE (2013)
44. Milić, M., Makajić-Nikolić, D.: Optimization of the cell-based software architecture by applying the community detection approach. In: 2024 19th Conference on Computer Science and Intelligence Systems (FedCSIS). pp. 149–156. IEEE (2024)
45. Millett, S., Tune, N.: Patterns, principles, and practices of domain-driven design. John Wiley & Sons (2015)
46. Mohamed, E.M., Agouti, T., Tikniouine, A., El Adnani, M.: A comprehensive literature review on community detection: Approaches and applications. Procedia Computer Science 151, 295–302 (2019)
47. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Physical review E 69(26113), 1–16 (2004)
48. Nikolić, L., Dimitrieski, V., Čeliković, M.: An approach for supporting transparent acid transactions over heterogeneous data stores in microservice architectures. Computer Science and Information Systems 21(1), 167–202 (2024)
49. Pan, W.F., Jiang, B., Li, B.: Refactoring software packages via community detection in complex software networks. International Journal of Automation and Computing 10(2), 157–166 (2013)
50. Rabinovich, S.G.: Measurement errors and uncertainties: theory and practice. Springer Science & Business Media (2005)
51. Rashid, J., Mahmood, T., Nisar, M.W.: A study on software metrics and its impact on software quality. arXiv preprint arXiv:1905.12922 (2019)
52. Roa, Y.N.: Taking advantage of cell-based architectures to build resilient and fault-tolerant systems. In: Gancarz, R. (ed.) Cell-Based Architectures: How to Build Scalable and Resilient Systems. No. 113, InfoQ eMag (2024), https://www.infoq.com/minibooks/cell-based-architecture-2024/, accessed: March 26, 2025
53. Sellami, K., Saied, M.A., Ouni, A., Abdalkareem, R.: Combining static and dynamic analysis to decompose monolithic application into microservices. In: International Conference on Service-Oriented Computing. pp. 203–218. Springer (2022)
54. Serrano, B., Vidal, T.: Community detection in the stochastic block model by mixed integer programming. Pattern Recognition 152, 110487 (2024)
55. Services, A.W.: Reducing the scope of impact with cell-based architecture: Aws well-architected. Available online: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html (2024), accessed: 2024-07-12
56. Sewak, M., Singh, S.: Winning in the era of serverless computing and function as a service. In: 2018 3rd International Conference for Convergence in Technology (I2CT). pp. 1–5. IEEE (2018)
57. Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R.: Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: 2020 USENIX annual technical conference (USENIX ATC 20). pp. 205–218 (2020)
58. Singhal, N., Sakthivel, U., Raj, P.: Selection mechanism of micro-services orchestration vs. choreography. International Journal of Web & Semantic Technology (IJWesT) 10(1), 25 (2019)
59. Srinivasan, K., Devi, T.: Software metrics validation methodologies in software engineering. International Journal of Software Engineering & Applications 5(6), 87 (2014)

60. Šubelj, L., Bajec, M.: Community structure of complex software systems: Analysis and applications. Physica A: Statistical Mechanics and its Applications 390(16), 2968–2975 (2011)
61. Venters, C.C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., Nakagawa, E.Y., Becker, C., Carrillo, C.: Software sustainability: Research and practice from a software architecture viewpoint. Journal of Systems and Software 138, 174–188 (2018)
62. Verdecchia, R., Kruchten, P., Lago, P., Malavolta, I.: Building and evaluating a theory of architectural technical debt in software-intensive systems. Journal of Systems and Software 176, 110925 (2021)
63. Viljoen, N.M., Joubert, J.W.: Supply chain micro-communities in urban areas. Journal of Transport Geography 74, 211–222 (2019)
64. Wang, Q., Kanemasa, Y., Li, J., Jayasinghe, D., Kawaba, M., Pu, C.: Response time reliability in cloud environments: an empirical study of n-tier applications at high resource utilization. In: 2012 IEEE 31st Symposium on Reliable Distributed Systems. pp. 378–383. IEEE (2012)

Appendix

Data Availability Statement. All relevant artifacts, including source code, heuristics, input data, and optimization results, are available at the following GitHub repository: https://github.com/milicm/optimizing-cell-based-software-architecture-community-detection.

Table 7. Effective interactions between software elements and software metric values after simulation

Metrics	Interactions								
	order -> legacy	payment -> notification	inventory -> production	product -> production	order -> notification	production -> product	production -> legacy	order -> product	production -> inventory
NoInt	15372	11322	67469	64425	7356	27886	33123	62122	47783
Total_ReqT	527979	1030516	616175	612443	694224	392293	511689	699353	596878
Total_ResT	22342	26597	19455	18254	12284	10053	12714	23151	14619
Total_ReqS	5053032	3902588	11267323	10758975	2462557	4835305	5862807	10523413	8362025
Total_ResS	5818935	3446692	11654157	11128212	1366186	8212586	50771373	18301326	13596598
Avg_ReqT	34.3467	91.0189	9.1327	9.5062	94.3752	14.0677	15.4481	11.2577	12.4914
Avg_ResT	1.45342	2.34914	0.28835	0.28334	1.66993	0.36050	0.38384	0.37267	0.30594
Norm_NoInt	0.13334	0.06597	1	0.94936	0	0.34152	0.42864	0.911054	0.67252
Norm_Avg_ReqT	0.29579	0.96063	0	0.00438	1	0.05789	0.07409	0.02493	0.039402
Norm_Avg_ResT	0.56640	1	0.00243	0	0.67121	0.03735	0.04865	0.04324	0.01094
Norm_Total_ReqS	0.29421	0.16355	1	0.94226	0	0.26948	0.38618	0.91551	0.67003
Norm_Total_ResS	0.09013	0.04211	0.20824	0.19759	0	0.13858	1	0.34278	0.24755
Communication Intensity (b_{ij})	0.276	0.4465	0.4421	0.4187	0.3342	0.169	0.3875	0.4475	0.3281

Table 8. First eight scenarios for the examination of the influence of weighting schemes on the community structure

Edges	equal	w1=1	w2=1	w3=1	w4=1	w5=1	w2,w3=0.5	w4,w5=0.5
(5, 7)	0.554	0.625	0.675	0.580	0.755	0.133	0.628	0.444
(7, 8)	0.415	1.000	0.116	0.167	0.099	0.695	0.141	0.397
(8, 4)	0.389	0.060	0.107	0.942	0.722	0.115	0.525	0.418
(4, 11)	0.741	0.999	0.681	0.946	0.600	0.477	0.814	0.539
(11, 2)	0.464	0.794	0.385	0.274	0.500	0.368	0.330	0.434
(2, 1)	0.558	0.714	0.555	0.933	0.196	0.393	0.744	0.295
(1, 5)	0.425	0.938	0.351	0.134	0.090	0.612	0.242	0.351
(8, 7)	0.704	0.251	0.807	0.980	0.754	0.729	0.893	0.742
(5, 1)	0.663	0.898	0.972	0.938	0.100	0.407	0.955	0.254
(6, 9)	0.487	0.205	0.775	0.830	0.415	0.210	0.803	0.313
(9, 10)	0.639	0.524	0.404	0.411	0.934	0.920	0.408	0.927
(10, 3)	0.503	0.693	0.181	0.427	0.575	0.638	0.304	0.606
(3, 12)	0.564	0.949	0.398	0.972	0.080	0.424	0.685	0.252
(12, 6)	0.649	0.759	0.680	0.543	0.378	0.884	0.611	0.631
(10, 9)	0.358	0.183	0.151	0.058	0.516	0.884	0.104	0.700
(9, 6)	0.611	0.812	0.815	0.306	0.189	0.933	0.561	0.561
(15, 16)	0.294	0.153	0.605	0.345	0.121	0.247	0.475	0.184
(16, 17)	0.280	0.175	0.266	0.288	0.376	0.297	0.277	0.336
(17, 13)	0.554	0.066	0.567	0.988	0.883	0.265	0.778	0.574
(13, 20)	0.361	0.153	0.518	0.478	0.154	0.504	0.498	0.329
(20, 19)	0.535	0.473	0.973	0.948	0.091	0.190	0.960	0.140
(19, 14)	0.494	0.346	0.626	0.713	0.655	0.128	0.670	0.392
(14, 15)	0.587	0.925	0.398	0.946	0.531	0.136	0.672	0.334
(20, 13)	0.761	0.918	0.691	0.795	0.969	0.431	0.743	0.700
(15, 14)	0.525	0.634	0.365	0.572	0.673	0.379	0.469	0.526
(18, 21)	0.641	0.106	0.987	0.830	0.902	0.379	0.908	0.641
(21, 18)	0.485	0.488	0.035	0.517	0.414	0.973	0.276	0.694
(23, 26)	0.385	0.778	0.205	0.005	0.253	0.683	0.105	0.468
(26, 28)	0.638	0.546	0.973	0.271	0.818	0.583	0.622	0.701
(28, 24)	0.767	0.916	0.591	0.924	0.584	0.819	0.757	0.702
(24, 30)	0.273	0.049	0.513	0.142	0.074	0.587	0.327	0.331
(30, 29)	0.272	0.238	0.469	0.095	0.238	0.323	0.282	0.280
(29, 22)	0.750	0.756	0.584	0.762	0.833	0.812	0.673	0.823
(22, 23)	0.617	0.673	0.103	0.759	0.797	0.754	0.431	0.775
(30, 24)	0.525	0.989	0.558	0.346	0.404	0.326	0.452	0.365
(22, 29)	0.463	0.759	0.235	0.289	0.541	0.491	0.262	0.516
(25, 27)	0.696	0.407	0.740	0.587	0.940	0.805	0.663	0.872
(27, 25)	0.539	0.599	0.318	0.782	0.003	0.996	0.550	0.500

Table 9. Last seven scenarios for the examination of the influence of weighting schemes on the community structure

Edges	w1=0	w2=0	w3=0	w4=0	w5=0	w2,w3=0	w4,w5=0
(5, 7)	0.373	0.537	0.441	0.238	0.598	0.501	0.606
(7, 8)	0.418	0.615	0.333	0.736	0.528	0.464	0.707
(8, 4)	0.403	0.482	0.555	0.615	0.703	0.631	0.452
(4, 11)	0.407	0.452	0.565	0.393	0.492	0.434	0.700
(11, 2)	0.431	0.548	0.596	0.531	0.859	0.726	0.331
(2, 1)	0.809	0.662	0.769	0.285	0.576	0.539	0.301
(1, 5)	0.705	0.686	0.370	0.492	0.471	0.343	0.574
(8, 7)	0.501	0.557	0.356	0.542	0.541	0.563	0.331
(5, 1)	0.368	0.542	0.494	0.707	0.166	0.291	0.590
(6, 9)	0.547	0.488	0.715	0.791	0.242	0.641	0.489
(9, 10)	0.260	0.474	0.340	0.668	0.458	0.432	0.487
(10, 3)	0.406	0.428	0.422	0.236	0.238	0.220	0.478
(3, 12)	0.256	0.478	0.367	0.466	0.348	0.814	0.732
(12, 6)	0.522	0.588	0.364	0.526	0.448	0.539	0.416
(10, 9)	0.323	0.752	0.499	0.393	0.346	0.633	0.509
(9, 6)	0.727	0.452	0.485	0.605	0.839	0.462	0.598
(15, 16)	0.473	0.512	0.450	0.306	0.771	0.452	0.906
(16, 17)	0.431	0.466	0.420	0.396	0.364	0.680	0.549
(17, 13)	0.423	0.410	0.543	0.343	0.738	0.400	0.531
(13, 20)	0.179	0.805	0.475	0.546	0.535	0.347	0.237
(20, 19)	0.207	0.121	0.812	0.349	0.652	0.491	0.686
(19, 14)	0.504	0.384	0.356	0.632	0.548	0.745	0.376
(14, 15)	0.663	0.699	0.508	0.315	0.470	0.715	0.423
(20, 13)	0.328	0.396	0.381	0.361	0.610	0.342	0.655
(15, 14)	0.472	0.417	0.504	0.700	0.409	0.461	0.435
(18, 21)	0.393	0.530	0.548	0.676	0.310	0.641	0.469
(21, 18)	0.405	0.440	0.617	0.406	0.630	0.593	0.617
(23, 26)	0.719	0.590	0.477	0.537	0.765	0.264	0.566
(26, 28)	0.601	0.747	0.535	0.237	0.481	0.481	0.777
(28, 24)	0.659	0.569	0.313	0.594	0.461	0.483	0.465
(24, 30)	0.352	0.346	0.641	0.484	0.584	0.880	0.689
(30, 29)	0.631	0.555	0.500	0.455	0.528	0.507	0.497
(29, 22)	0.513	0.584	0.177	0.324	0.340	0.134	0.670
(22, 23)	0.520	0.627	0.266	0.347	0.686	0.553	0.802
(30, 24)	0.475	0.444	0.639	0.587	0.329	0.597	0.694
(22, 29)	0.617	0.641	0.434	0.457	0.384	0.091	0.587
(25, 27)	0.304	0.550	0.482	0.446	0.759	0.605	0.563
(27, 25)	0.380	0.429	0.344	0.674	0.374	0.891	0.665

Miloš Milić is an Associate Professor at the University of Belgrade, Faculty of Organizational Sciences, Department of Software Engineering. His research and professional interests include software quality, software design, software architectures, design patterns, software engineering technologies, and software engineering education. He teaches undergraduate and graduate courses in his field. Miloš Milić serves as the President of the Committee for Standards and Related Documents in the fields of Software Engineering, IT in Education, and the Internet at the Institute for Standardization of Serbia.

Nebojša Nikolić is an Associate Professor at the University of Belgrade, Faculty of Organizational Sciences, Department of Mathematics. His research interests include applied mathematics, graph theory, combinatorial optimization, and metaheuristics. He teaches undergraduate courses in mathematics and PhD-level courses in combinatorial optimization and combinatorial algorithms. Nebojša Nikolić is the head of the Department of Mathematics and a member of the Mathematical Society of Serbia.

Dragana Makajić-Nikolić is a Full Professor at the University of Belgrade, Faculty of Organizational Sciences, Department of Operations Research and Statistics. Her research interests are related to mathematical modelling, optimization methods, business analytics, reliability, and risk assessment. She is the author or co-author of over 150 papers in international and national journals and conferences, several chapters in international monographs and encyclopaedias, and three books in the OR field. Dragana Makajić-Nikolić is the head of DOPIS – the Serbian Operational Research Society, and a member of UETS – the Union of Engineers and Technicians of Serbia, and the International Association of Engineers (IAENG).

Received: March 30, 2025; Accepted: November 10, 2025.