# An Innovative Metric-based Clustering Approach for Increased Scalability and Dependency Elimination in Monolithic Legacy Systems

**Abdulaziz Aljaloud**
College of Computer Science & Engineering, University of Ha'il, Saudi Arabia
a.aljaloud@uoh.edu.sa

**Abdul Razzaq**
Ocean Technology and Engineering, Ocean College, Zhejiang University, PR of China
11934071@zju.edu.cn

*Received: 16 May 2023 | Revised: 30 May 2023 | Accepted: 8 June 2023*

 *DOI: https://doi.org/10.48084/etasr.6048*

**ABSTRACT**

**Scalability is one of the system's characteristics highlighted in the recent literature, and it is directly related to issues that are encountered in state-of-the-practice technology. The scalability of a system is challenging because monolithic legacy systems are hard to scale due to the high level of component dependencies. To the best of our knowledge, there is no published work available that can identify the components from a monolithic legacy system in the context of dependent and independent components and scale them accordingly. The main contribution of this paper is the proposal of a novel approach for the exclusive identification of dependent and independent monolithic legacy system components. The proposed approach also helps to remove the dependency among components of monolithic legacy systems. As a result, it establishes a precise method that identifies all the components of an application and removes the dependency among components, helping to increase the scalability of the resulting application. This approach was validated by several experiments, and the key findings were the identification of dependent and independent components, the identification of relationships among components, and the identification of the abstract level architecture of the monolithic legacy system. In future work, the proposed method will be enhanced toward the recovery of the whole system's architecture.**

***Keywords-monolithic application upgradation; system components; architecture recovery; dependency; scalability of monolithic systems; reverse engineering***

## I. INTRODUCTION

Scalability is a needed quality attribute of a process or system of non-functional requirements that describes the ability of the software to scale analogous to the demands (load scalability, structural scalability) [1-5]. Scalability saves resources in terms of the time required to scale the software [6]. If scalability issues are not fixed rapidly and timely, the system can become unmaintainable [5]. At the point when scalability isn't altogether disregarded, requirements regularly are described distinctly regarding application boundaries.

- Load scalability: A component of a monolithic legacy system can be added, removed, or modified to accommodate changing loads. The system has the ability to adapt graciously as the presented circulation increases [4].
- Structural scalability is the capacity of a system to extend in a picked measurement without significant modifications to its architecture, and its usage or models don't impede the development of the number of objects it encompasses [4].

A monolithic legacy system architecture is the traditional unified model for software development and design. Traditional software is intended to be self-contained. Software components are interconnected and related as opposed to loosely coupled just like the case with a software modular [7]. This can create many challenges for groups working in a similar environment [8]. A monolithic legacy system is simple to develop and to test and has numerous obligations, e.g. it is independent from other applications and self-contained. These traditional architectures are commonly hard to upgrade, deploy, and maintain, and difficult to understand [9].

After the evaluation of the relevant research, it has been established that dependency removal of a component to increase the scalability of a system is an area where a serious effort is required to bridge the identified knowledge gaps.

### *A. Components*

Components play a significant role in reducing the scalability issues of monolithic legacy systems [10]. Components are defined as the smallest self-managing, autonomous, and helpful sets of a system that works in various environments. Components are regularly circulated objects consolidating propelled self-administration features [11]. The components utilized in these essential programming applications are comprised of central blocks that can be joined together, relying on the prerequisite [12]. Components increase the productivity of application developers and improve the programming quality due to the high level of reusability [13]. It is vital for an application developer to alter parts since it is uncommon to discover the components coordinating their non-functional and functional requirements in new system applications [13]. Software engineering is favored due to these reasons. It can address these worries beginning from the prerequisite of the undertaking. The activities are to be executed by the product of a specific business for which it is planned [14]. Additionally, it can likewise guarantee the duty regarding individual groups. Software development based on components has emerged as a successful way to deal with building an adaptable system. Component-based work has risen as a compelling way to deal with complex programming systems [15]. Its advantages incorporate decreased improvement costs through reusing off-the-shelf components and expanded flexibility through including, evacuating, or replacing components [16, 17].

The system applications are to be kept running in the cloud with effectiveness. This requires substantially more expertise than what is essential to convey any programming in virtual machines. It is continuously prescribed to oversee cloud applications consistently to use their assets as per the approaching burden and to confront the disappointments, to duplicate and rehash every one of the segments to give flexibility if there should arise an occurrence of inconsistent framework [18]. When a program is planned to keep in view every one of the prerequisites, it turns out to be very extreme for the software architect to present radical changes which are later on requested by plans of action or clients as often as possible since it turns out to be progressively entangling for the designer to make changes when the code begins extending as a result of the inclusion of various individuals who make changes in the product [19]. As increasingly more exertion is required to facilitate updating in the highly coupled architecture of a monolithic design. This entire procedure, makes the discharge cycle of the application moderate [20] and the model delicate and untrustworthy. Versatility is an essential element that requires the task and advancement of large enterprise applications [21]. The major downside of the monolithic application is its deficiency of scalability when a specific errand is to be performed inside the components [22]. The lengthy software cycle in light of the multifaceted nature of the framework is an obstacle in current, dependable administrations. In this strategy, the figuring method delivers the wanted outcomes. The method utilized is bunching, which is considered the least essential and challenging procedure utilized in building and science [12]. The primary and most essential target of executing this system is to mention the objective facts clearer to build up a superior comprehension. This robust understanding makes it simple to create complex information structures from given directives. A grouping strategy or technique is commonly used to distinguish all the related segments of monolithic legacy systems alongside their duties. As the info utilized in this procedure features the interconnectivity of every one of these parts, this grouping strategy is beneficial to limit the interconnection among various components to create ideal outcomes.

### *B. Clustering*

In the proposed method, reverse engineering is used to produce the wanted outcome. This method utilized with the end goal of reverse engineering is considered the least complicated and the primary method utilized in engineering and science [12, 23]. The primary and most imperative target of implementing this method is to mention the objective facts clearer in order to build up a superior understanding. This awareness makes it simple to create a sophisticated learning structure from given highlights. Bunching strategy or technique is commonly liked to recognize all the related parts of monolithic legacy systems alongside their obligations [24]. As the information utilized in this strategy features the interconnectivity of every one of these components, this metric-based clustering method is very valuable to limit the interconnection among various parts in order to create the ideal outcome [25, 26]. Clustering is a procedure in which huge frameworks are divided into pieces. This sensible framework exhibits that the substances which bear closeness with each other have a place in a similar subsystem, while the elements with a contrast among each other are ordered into various subsystems [12]. The clustering procedure is commonly utilized in distinguishing the product parts [18, 27]. A software developer with vast experience may highlight two kinds of issues in practice. The first issue is that it is challenging to decide on an explicit cluster, which is utilized for profoundly coupled parts [26, 28]. The second issue is to decide on the bunch mapping, which connects the software components [29].

### *C. Metrics*

Metrics are tools for gathering and organizing data into coherent groups. In the context of this particular research, metrics assume a crucial role in assessing the various components of a monolithic legacy system [32, 33]. By employing metrics, we are able to effectively categorize and cluster all relevant system components. Through this process, we discover the intricate relationships between these components, identifying dependencies and independencies among them. These metrics provide us with a comprehensive understanding of the system's inner working and enable us to gain insight into its overall structure and functionality. By mapping out the interdependencies, we are able to discern the impact that changes or modifications in one component could have on others, aiding us in making informed decisions regarding system maintenance, optimization, or potential refactoring efforts. Furthermore, the metrics serve as a means to quantify the system's performance, highlighting areas of strength and potential weaknesses or bottlenecks. This information is proved to be invaluable in the pursuit of improving a system's overall efficiency and reliability. Through

the diligent utilization of metrics, we are able to extract meaningful insight and enhance our comprehension of a monolithic legacy system, ultimately contributing to its enhancement and evolution.

### *D. Related Work*

There are many approaches in identifying the software components of monolithic legacy systems. Authors in [34] find the components based on classes and use a clustering algorithm. The approach presented in [35] makes use of only variables related directly to components. However, this can lead to low component quality. The identified components do not fulfill the requirements of scalability. Authors in [34] relied extensively on UML diagrams only, which may not mirror the original structure of a system [36, 37]. However, our method is general and based on the investigation of the source code [38], which leaves the leading software design curio that mirrors the truth of a system. To the best of our knowledge, there is not a method/technique in the literature for identifying the components of a monolithic legacy system in the context of dependent and independent components and relationships among them. It is necessary to scale the developed monolithic legacy system. By using the proposed method, we identified the components and also found the relations among them and made an abstract-level architecture of the monolithic legacy system. In this way, software architecture, open-source advancement, authoritative structure, and obligation are vertically disintegrated [27]. During the time spent on software development, a complex system is challenging to be architected expertly. So, architecture refinement plays a critical job in the description of software architecture [12]. In a stepwise refinement, a succession of steps beginning from a unique detail of the design prompts a reliable, execution-focused, and building model [39]. To the point of a component, the conceptual architecture could be refined to a progressively robust design by sets. The architecture comprises two parts and a connector. The reality of software architecture representation gives numerous points of interest amid all the periods of the programming life cycle [20, 40]. For some systems, like the inheritance ones, there is no available representation of their architecture [41, 42]. The interface of a component should be the essential concern of its designer or developer. Since the components are intended for use in an assortment of systems and need to give the administrations the ability to pay a little heed to the setting, designers endeavoring to utilize a segment must almost certainly distinguish the capacity of a component and the methods for invoking its behavior [43]. Monolithic applications come up with failures when the number of clients getting to a system becomes excessively high or when too many features are integrated into a single system [20]. Component architecture gives software engineers a way to deal with the multifaceted nature of vast-scale logical recreations and to push toward a fitting and-play condition for elite figuring [44]. Some vast and expensive software systems work in a constant domain under requesting execution prerequisites. Often a large portion of the expense for an item is spent on support [45]. Numerous advantages can be picked up by partitioning a system into components. Additionally, maintainability and scalability are accomplished [46, 47]. A comparison of some appropriate methods is shown in Table I.

TABLE I. COMPARISON OF PREVIOUS AND PRESENT WORK

| Previous works | Present work |
|---|---|
| -Identification of only components<br>-No identification of dependent components & relationships among components<br>-No identification of the architecture of the system | -Identification of independent components<br>-Identification of dependent components<br>-Identification of relationships among the components<br>-Identification of abstract level architecture of the monolithic legacy system |

### *E. Contribution of this Study*

- Method Creation: A method was developed to identify dependent and independent components in a monolithic legacy system using metrics and clustering techniques.
- Scalability Enhancement: The method was successfully applied in a multinational industrial project, resulting in increased scalability.
- Dependency Removal: The dependent and independent components were identified and dependencies among the dependent components were eliminated by adding a few lines of code.
- Abstract-Level Architecture: An abstract-level architecture derived from the monolithic legacy system was achieved, providing higher-level insight into the system's design.
- Solution Applicability: Considering the applicability of the solution in terms of methods, dependencies, and architecture level refactoring, the solution needs to be further evaluated in terms of legacy migration for emerging software. Specifically, we focused on evaluating refactoring or modernization of the existing software to modern computing platforms such as mobile computing [43] and architecture-level refactoring of quantum software [44].
- Featured Method: This research is an effort to fill the identified gap in the literature. Scaling a monolithic legacy system can be useful for academic and industrial works. The monolithic legacy system can be scaled after its components are identified. It is useful for monolithic legacy systems that have no documentation and their components are unknown, to know how can the system be scaled further. This method helps achieve high cohesion and low coupling of the monolithic legacy system's components.

## II. RESEARCH METHOD AND COMPONENT MAPPING

### *A. Research Method*

#### *1) Research Questions (RQs)*

- RQ1. What techniques/methods are reported in the literature regarding the identification of the components of the developed monolithic legacy system and the removal of the dependencies of applications' components?
- RQ2. Why is it necessary to find and remove the dependency on the monolithic legacy systems application's components?

- RQ3. How to resolve the dependency issue of the application's components in order to increase the scalability of the monolithic legacy system?
- RQ4. What is the impact of the novel proposal when compared with the state of the art?

TABLE II. RQs DISCUSSION

| RQs | Discussion |
|---|---|
| **RQ1** | We did not find any method or technique for finding the components of developed systems and for removing component dependencies. |
| **RQ2** | Scalability is the primary quality attribute to achieve by removing the dependencies of the application's components. |
| **RQ3** | We developed the metrics-based clustering method to identify the dependent and independent components of the applications and added a few new lines of code to remove dependencies. |
| **RQ4** | We identified the conceptual architecture of the monolithic legacy system and identified the relations among all components of the application. It's abstract level architecture. |

The research method comprises the following components:

- Critical literature evaluation and analysis.
- A method for proof of concept is built in order to validate results.
- Case study results and analysis of data for answering the raised research questions.
- Concluding the research based on analyzed data and own interpretive deductions.

*2) Rationale*

Table II discusses the research questions. We undertook the study in order to find the dependent and independent components of a monolithic legacy system. After finding all the components of the system, we removed the dependencies of components and classes by using the proposed model with metrics-based clustering. The monolithic legacy system can scale by using the proposed method.

*3) Type of Study*

The contextual investigation is similar in nature to what will be utilized in the current study. Comparative study is conducted to find out the impact of large and small projects in order to remove the component dependencies.

*4) Study Analysis*

We measured and compared the results of the project keeping in mind the reusability of components and the achievement parts (e.g. dependent and independent components, relationships among components, and abstract level system's architecture) of the proposed model.

*5) Case Study Context*

Proposing a new technique is the primary context of the case study.

*6) Expected Result*

By using the proposed method for increasing the scalability of a monolithic legacy system, the expected result was completed successfully.

*7) Component Mapping*

To identify the software components of monolithic legacy systems, we made the component model of Figure 2 and showed the relationships among all methods (Figure 1).

### *B. Attribute Relations*

Attributes are variables where data are stored temporarily. The first step of the proposed method is to start from the variables in order to find the dependent and independent components of the monolithic legacy system. We provide a list of attributes in Figure 1 (V1-13, V1-V59, V60-V67). A = Attribute, V= Variable

### *C. Methods*

In the second step, we list down all the methods (Table III). We identified the relationships between methods and attributes (Figure 1).

### *D. Attributes to Methods*

We listed down all the attributes and created relevant groups. We define the groups of methods with relevant attributes (Figure 1).

### *E. Mapping Methods to Component*

The different method collections are defined (Figure 1). Each silhouette is composed of different sets of methods. The silhouette interface is the boundary of methods and sets of attributes inside the silhouette center. These interfaces have a link with other methods from the outside of the silhouette. Figure 2 describes the attribute mapping component model. This model shows the mapping structure of the components.

### *F. Lack of Cohesion of Methods (LCOM)*

The single obligation rule expresses that a class should not have more than one motivation to change. Such a class is said to be durable. A high LCOM esteem, by and large, pinpoints an inadequately cohesive class.

LCOM: max ((1-2), 0) (1)

Maximal Cohesion: attributes are accessed by all methods LCOM = 0 (2)

No cohesion: a unique attribute is accessed by each method LCOM = 1 (3)

$$LCOM(C) = \begin{cases} J - Q & \text{if } J > Q \\ 0 & \text{Otherwise} \end{cases} \quad (4)$$

where $J$ is the number of pairs of discrete strategies in $C$ which don't share instance attributes, $Q$ is the number of pairs of discrete strategies in $C$ which share the instance attributes, $m$ is the number of methods, $a$ the number of attributes, $m(A_i)$ the number of methods that access $A_i$

$$m(A_i) = \frac{\left(\frac{1}{a}\sum_{i=1}^{a} m(A_i)\right) - m}{1 - m} \quad (5)$$

Maximal Cohesion: all methods access all attributes

$m(A_i) = m$ and LCOM = 0 (6)

No cohesion: each method accesses a unique attribute

$m(A_i) = 1$ and LCOM = 1 (7)

If some attributes are not accessed at all, then:

$$m(A_i) = 0 \quad (8)$$

and if no attributes are accessed:

$$\frac{\left(\frac{1}{a}\sum_{i=1}^{a} m(A_i)\right)-m}{1-m} = \frac{-m}{1-m} = 1 + \frac{1}{m-1} \quad (9)$$

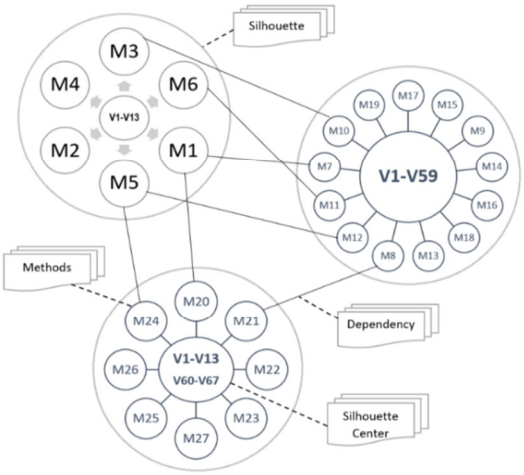


Fig. 1. Method relationships and structure.

## III. PROPOSED SOLUTION MODEL

### A. Solution Brief

In this proposed method, we used different techniques (clustering, matrices, and LCOM matrix formula). This approach is divided into different categories. First, we extract the abstract-level solution model for understanding the method flow and its different sections (Figure 1) in an abstract-level model design. After this, we made a detailed solution design model with different sub-phases, which shows the holistic flow of the solution design.

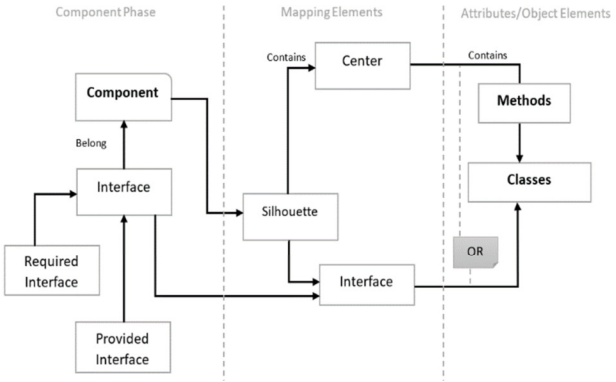


Fig. 2. Attribute and object mapping component model.

### B. Project Brief

The monolithic legacy system is a .net project. There were two projects, the first one a small pilot study, and the second is one of medium level. The proposed technique was applied to both projects. We measured the results and the type of impact on through this proposed method. We used the proposed method on classes and methods/functions to find the similarities and dependencies of every class and method/function. This helped us to recognize the components of the monolithic legacy system. To identify the monolithic legacy system components, two sets of categories of components were made, independent components and dependent components. The way to verify dependent components is mentioned below. Finally, we removed the component dependencies (Figure 1) and scaled the monolithic legacy system by adding a new component. The medium level project development was made in the MVC .NET Framework. The duration of this project was four months and two developers were employed. The number of code lines was 7,944 and a total of 330 classes formed the depth of inheritance. Further details and significant parts of the project are mentioned in Table III.

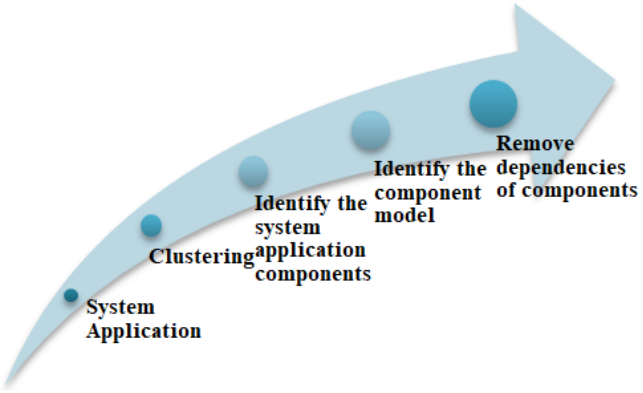


Fig. 3. Abstract model of the proposed solution.

TABLE III. PROJECT CODE DETAILS

| Parts (Operations) | Class coupling | LOC |
|---|---|---|
| Areas | | |
| Admin section | 7 | 29 |
| Admin controller | 82 | 1117 |
| Admin models | 95 | 1906 |
| Approval | 4 | 4 |
| Approval controllers | 57 | 912 |
| Budget | 4 | 4 |
| Budget controllers | 2 | 7 |
| Budget models | 17 | 71 |
| Request | 4 | 4 |
| Request controllers | 29 | 109 |
| Request models | 24 | 217 |
| User management | 4 | 4 |
| User management controllers | 117 | 1638 |
| User management models | 70 | 807 |
| Vendor | 4 | 4 |
| Vendor controllers | 35 | 170 |
| Vendor models | 31 | 213 |

Figure 3 shows the procedure at the abstract level of our proposed method and its flow. The proposed method has five processes. The first process is the project. Projects must have classes or methods/functions. The second process is the clustering technique, which we used with metrics in our method. The third process is identifying components by applying the proposed method. The fourth process gives us an

abstract-level architecture model of component relations. In the last process, we identify dependent and independent components. We add a few lines of code to remove the dependencies where it is necessary.

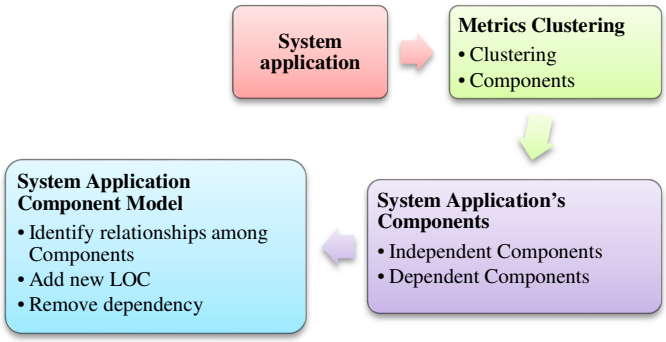


Fig. 4. Holistic proposed solution model.

The project in the first process is a monolithic legacy system that can be a web application or desktop software and can be written in any programing language. The second process is the basis of the proposed method, which we designed using the clustering technique. The proposed method is based on metrics. We used clustering techniques in our method for making groups or related methods that are being used in a monolithic legacy system. We find the code methods/functions and insert each method/function in the metric. We also write the value of each method, i.e. how many times this method/function is used in other methods. Also, we checked and wrote the accessibility of each method for another method. We counted the number of usage values by applying intersection and writing down the exact values in the metric and how many times it is being used in other methods. The third process is defining a monolithic legacy system's components. In this process, we find components based on our metrics, check the relations, and make groups by using the clustering technique and further separate them into groups. Each group expresses its related component. We measured the relationships among methods/functions and found the relations among components that show the dependency among them. The independent components were also found. The fourth process is a component architecture model of the monolithic legacy system. To reach this process, we have identified the dependent and independent components of the monolithic legacy system and the relationships among all components. We see the complexity of the methods and the complexity of the components' relationships to make them independent. We make new classes or write code for methods to make them independent. We make the abstract-level architecture model of components by using their relationships. It is a high-level architecture model.

## IV. RESULTS AND DISCUSSION

We performed experiments on an industry-based monolithic legacy system and a pilot study. We found that the impact was the same in both studies. Our approach helps the developer identify components and the system's structure. It also allows the architect to create abstract-level architecture. We performed the clustering technique with metrics on components to find the similarity and dependency of each method used in the monolithic legacy system. Based on the proposed method, we identified the monolithic legacy system's components in two different categories of components, independent, and dependent. Independent components are easily scaled and reused without any risk, but dependent components cannot be easily scaled or upgraded in the monolithic legacy system, so it was needed to make the dependent components independent. We also identified the abstract level architecture of the monolithic legacy system. Once we identified all the system's components, we created the architecture of the monolithic legacy system based on the components' relations and the method's logic (Figure 7).

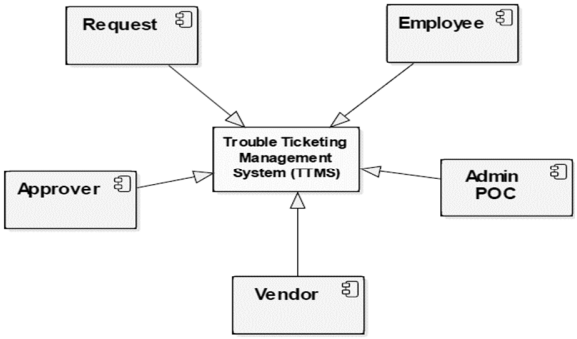


Fig. 5. TTMS component abstract architecture model.

Figure 5 shows all the components used in this project. These components are fully functional in the system. We designed all component models with the Enterprise Architect Tool. We used the standard rules, the "provider," and the "required" interfaces. Details about Figure 5 are described below:

- Request: it has the functionality to execute a request by Employee, Admin POC, or Approver. Requests can be generated under different conditions like priority, category, and location wise.
- Employee: it creates the request and checks its progress. Emails are generated on the biases of each request's action. An employee can communicate directly to its related POC based on the request. An employee can only create a request based on an auto-detected location, authenticated by an active directory (AD).
- Admin POC: it can create a request for itself and also on behalf of an employee. It has the capability to make the request on behalf of employees who are under the POC. Once a request is created on behalf of an employee, an email is sent to the employee about its update. The POC also has limited rights based on locations and category. The POC can only create or proceed with a request which is under the categories and locations' rights. The POC proceeds with the request that is created by the employee or itself. The POC can entertain any request which comes under its roles. It can reject the request made by the and

send a comment about the rejection whereas it can also communicate for further information about the request. After completion, of all proceedings of request and return from the vendor, the POC verifies the request and generates the invoice according to verification. All details of the invoice were automatically fetched from the database, which has been updated by the vendor. The POC can adjust the amount if something is missed by the vendor and let them know. This adjustment history is maintained with previous and new records. A POC can generate reports in detail or summary and with different filters to see things like categories, locations, and dates.

- **Approver**: it can proceed with the new request by itself. The approver can proceed with the request, which is forwarded by the POC for approval. The Approver can entertain any request amd submit it to the vendor, which comes under its roles and amount limit. The Approver can communicate with POC regarding requests or can reject the request back to POC with reasons. An Approver can check the request's details. The Approver also has the role of seeing the reports.
- **Vendor**: The Vendor must deliver the required items in requests. If a Vendor does not have a task, it can update the missing items in a request. The Vendor can communicate with the POC if it needs more details about the request's items. The Vendor can respond to the request and can check its progress. Once the job is done and all the items of the request are delivered, the Vendor closes the request as completed. If some items are delivered, the Vendor can update the request's status as a partial and can close it.

Figure 6 displays the intersection point between the Admin POC and the Approver components with the Employee component. This diagram visually represents the interdependencies that exist among these components, particularly their reliance on the Employee (request) component. The Admin POC and Approver components are intricately connected to the Employee component, as they rely on its functionalities to perform their respective tasks.

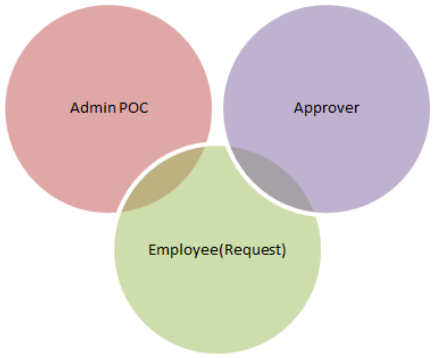


Fig. 6. Intersection among components.

Figure 7 provides a visual representation of the interfaces, requirements, and providers associated with each component within the system. It is important to note that Figure 7 represents the dependent component model prior to implementing the proposed method. Within this model, a clear observation is made: all components, namely Employee, Admin POC, and Approver do not depend on the Request component. This dependency indicates that the functionality and proper operation of these components rely on the availability and proper functioning of the Request component.

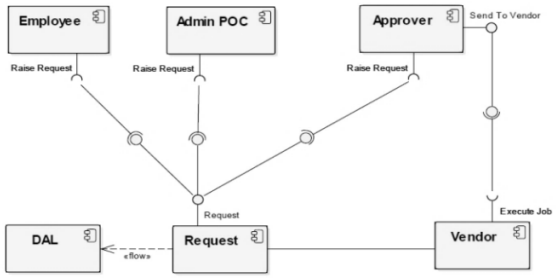


Fig. 7. TTMS component relation architecture design with required and provider interface.

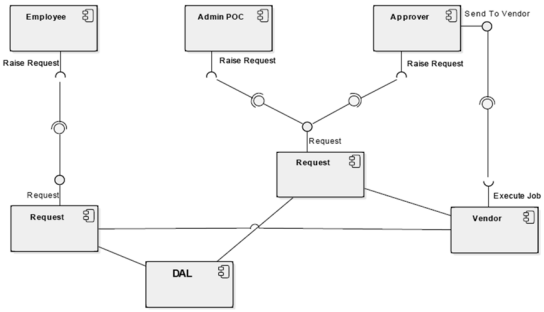


Fig. 8. Independent component model.

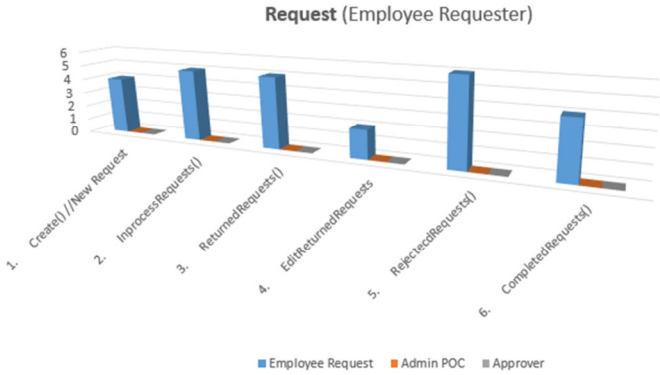


Fig. 9. Employee Request component.

Figure 8 shows the independence of the Requester component among all components. This component model is created after applying our proposed model, which clearly shows the elimination of component dependencies. Employee, Admin POC, and Approver components were dependent on the Request component before. Now, after removing the dependency, the employee component is separated from the Request component. Figure 9 presents a comprehensive overview of the dependencies within the system, showcasing the intricate interconnections among various methods and components. This diagram elucidates the profound bond

established with a specific method and how it affects the functioning of other components. Interdependencies with other components create a web of relationships that significantly impact the overall system's functionality and performance. This depiction of dependencies in Figure 9 reinforces the need for a comprehensive approach to system design and maintenance, where careful consideration is given to understanding and managing these intricate bonds. By recognizing the significance of these dependencies, system architects can make informed decisions to optimize system performance, enhance modularity, and enable seamless integration of future enhancements or modifications.

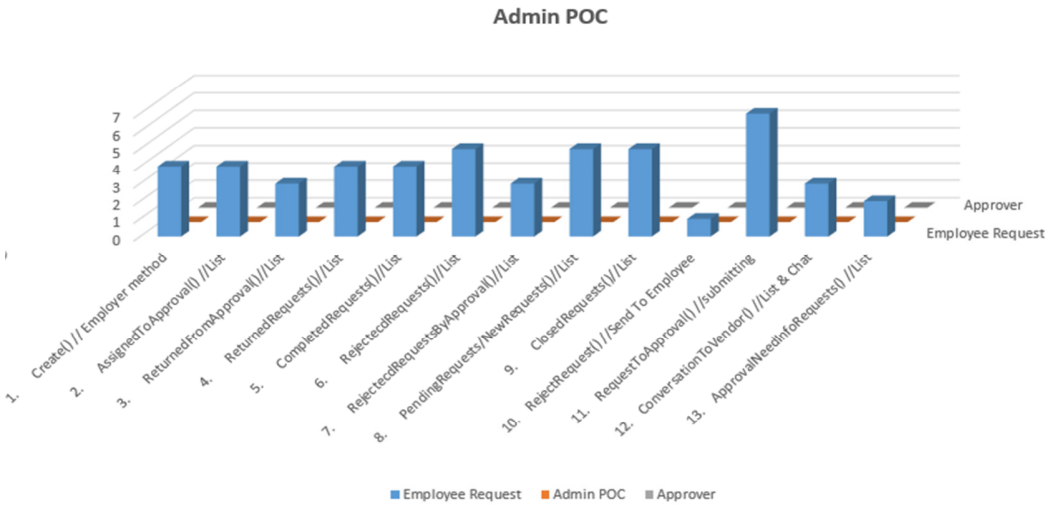


Fig. 10. Admin POC dependent component.

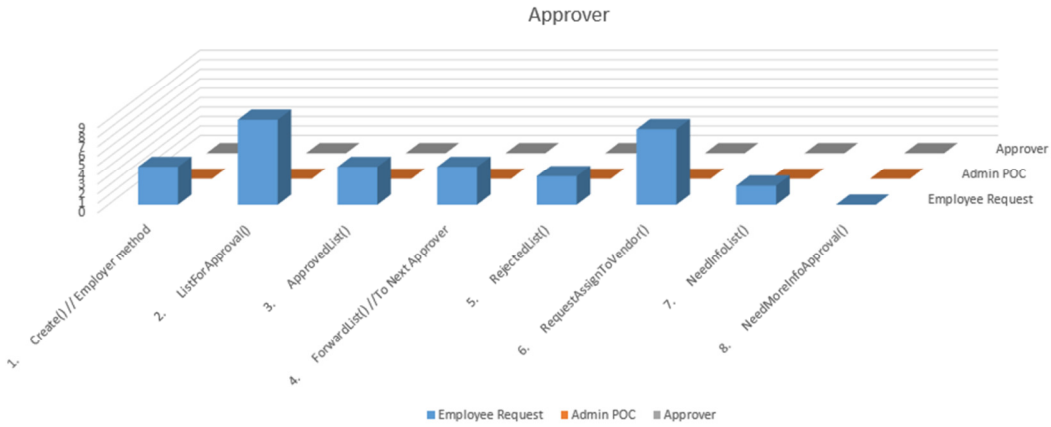


Fig. 11. Approver dependent component.

Figure 10 provides a clear visualization of the dependencies between all methods within the system and the Request component. This diagram highlights the crucial role of the Request component as a central entity, upon which various methods rely for their operations. It demonstrates the intricate network of dependencies, emphasizing that each method requires the functionality and data provided by the Request component to fulfill its purpose effectively. Moreover, it is worth noting that a significant change has been implemented concerning the dependency of the Admin POC component, as indicated in Figure 15. This change removes the dependency of the Admin POC component on the Request component, which was previously present. This alteration reflects a modification in the system's architecture, enabling the Admin POC component to operate independently, without relying on the Request component for its functionalities. By visualizing the dependencies in Figure 10, system designers and developers can gain valuable insight into the relationships among methods and components. It provides a basis for identifying potential areas of optimization, enhancing modularity, and improving the overall efficiency of the system. The removal of the dependency of the Admin POC component, as depicted in Figure 15, demonstrates a proactive approach to decoupling dependencies and refining the system's architecture for enhanced flexibility and adaptability.

Figure 11 visually illustrates the dependencies between all methods and the Approver component within the system. This diagram showcases the vital role of the Approver component as a central entity that various methods rely on for their functionality. It effectively portrays the intricate network of dependencies, emphasizing that each method requires the Request component to fulfill its specific tasks effectively.

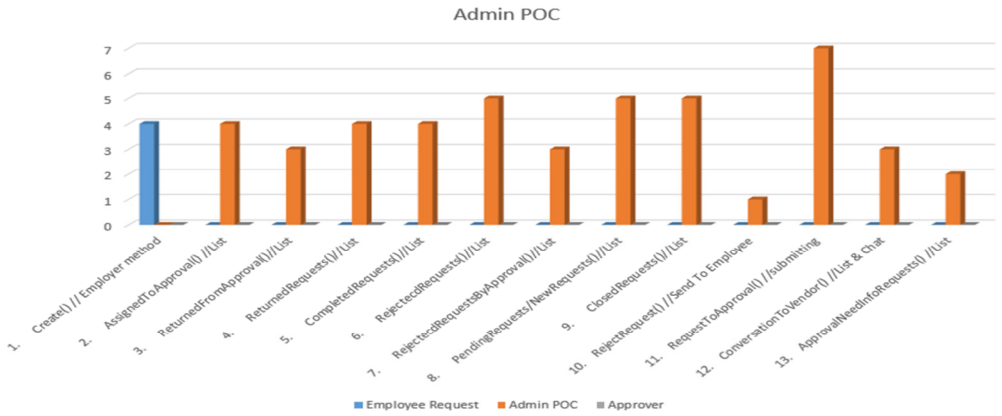


Fig. 12. Admin POC component dependent on Request creating method.

Figure 16 illustrates a significant modification made to the system's architecture, specifically the removal of the dependency on the Approver's component. This change, as depicted, enables the Approver's component to function independently, freeing it from relying on the Request component for its operations. The visual representation in Figure 11, combined with the adjustment showcased in Figure 16, emphasizes the system's adaptability and flexibility. It reflects the effort to reduce dependencies and enhance the modularity within the system's design. By removing the dependency on the Approver's component and accommodating different request models, the system becomes more versatile and capable of handling diverse scenarios effectively. These modifications not only improve the overall efficiency and maintainability of the system but also pave the way for future scalability and extensibility. System designers and developers can utilize the insights provided by Figures 11 and 16 to optimize the system's architecture, ensuring smooth operation and facilitating seamless integration of additional features or enhancements.

Table IV serves as a valuable qualitative representation of the dependency relationships among all components within the system. It provides insightful information about the ways different components rely on each other to fulfill their functionalities. By examining Table IV, we can observe that the Employee (Request) and Admin components depend on each other, implying that their operations are closely intertwined. Additionally, the Approver's method is primarily reliant on the Create() method, indicating a more specific dependency within the system. Table V showcases the transformative outcome after applying the proposed method and removing the dependencies among components. The Table clearly demonstrates that the monolithic legacy system's components have become independent, marking a significant shift in the system's architecture. Table V represents the main result of the decoupling process, highlighting the successful attainment of independent components that were previously dependent. The removal of dependencies has freed the components from relying on each other, enabling them to operate autonomously. This newfound independence among the components has substantial implications to the system. It enhances modularity, flexibility, and scalability, allowing for easier maintenance and future modifications. Each component can now be modified, updated, or replaced without causing disruptions to other components, fostering a more efficient and adaptable system. The results as presented in Table V, reinforces the success of the proposed method in decoupling the components and transforming the monolithic legacy system into a more independent and robust architecture. System designers can refer to this Table in order to understand the impact of the applied changes and assess the overall effectiveness of the method in achieving the desired outcome.

TABLE IV. DEPENDENT COMPONENTS OF THE MONOLITHIC LEGACY SYSTEM

| Components | Methods | Employee | Admin POC | Approver |
|---|---|---|---|---|
| **Employee Admin POC Approver** | Create() //New Request | Yes | Yes | Yes |
| | InprocessRequests() | Yes | Yes | No |
| | ReturnedRequests() | Yes | Yes | No |
| | EditReturnedRequests | Yes | Yes | No |
| | RejectecdRequests() | Yes | Yes | No |
| | CompletedRequests() | Yes | Yes | No |

TABLE V. INDEPENDENT COMPONENTS OF THE MONOLITHIC LEGACY SYSTEM

| Components | Methods | Employee | Admin POC | Approver |
|---|---|---|---|---|
| **Employee Admin POC Approver** | Create() //New Request | Yes | No | No |
| | InprocessRequests() | Yes | No | No |
| | ReturnedRequests() | Yes | No | No |
| | EditReturnedRequests | Yes | No | No |
| | RejectecdRequests() | Yes | No | No |
| | CompletedRequests() | Yes | No | No |

Figure 12 depicts the dependency relationship with a Requester component within the system. This diagram visualizes how other components rely on the requester component for certain functionalities. It highlights the integral role played by the Requester component in facilitating communication and data exchange within the system. The basis for this transformation is reflected in Table V, which likely served as a reference for identifying and resolving dependencies. By eliminating the dependency on the Requester component, the system achieves greater independence and modularity. This architectural change enables the requester component to function autonomously, without relying on external dependencies. Figure 15 represents the culmination of these efforts, illustrating the requester component as dependency-free. This revised graph demonstrates the success of the proposed method in removing dependencies and streamlining the system's architecture. The removal of dependencies offers several advantages, including enhanced flexibility, improved maintainability, and the ability to modify or update individual components without affecting the entire system's functionality.

By referring to Figures 12 and 15, system designers and developers can gain valuable insight into the evolution of the system's dependency structure and the resulting benefits of removing dependencies. These visual representations provide a clear understanding of the architectural improvements achieved through the proposed method, facilitating a more efficient and adaptable system design.

Figure 12 presents the dependency relationship between the Create() method and the Request component within the system. This diagram visually depicts how the Create() method relies on the functionality provided by the Request component. It highlights the integral role played by the Request component in facilitating the execution of the Create() method. The removal of dependencies between the Create() method and the Request component offers several benefits. It enhances the modularity and flexibility of the system, allowing for independent modifications and updates to each component without affecting the others. It also simplifies maintenance and reduces the risk of cascading failures that could arise from interdependencies. By referring to Figure 13 and Figure 16, system designers and developers can gain valuable insight into the evolution of the system's dependency structure. These visual representations effectively showcase the successful implementation of the proposed method in removing dependencies and optimizing the system's architecture.

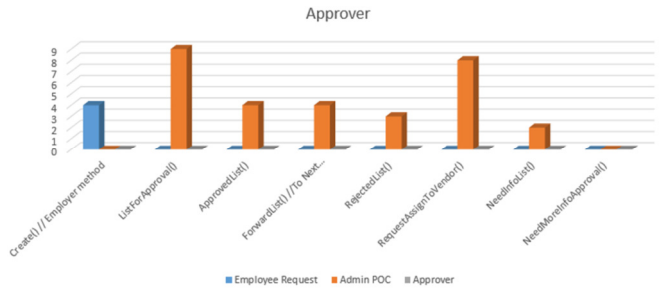


Fig. 13. Approver component dependent on the Request creating method.

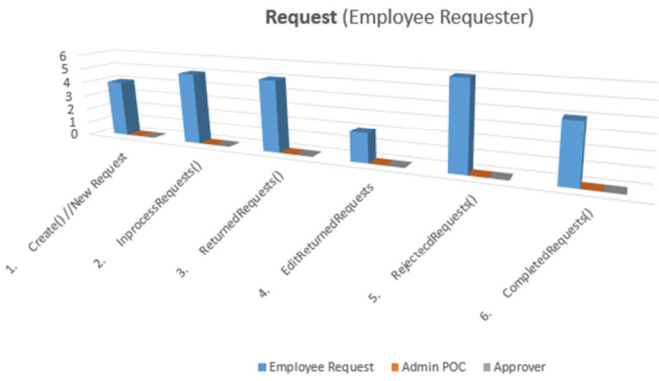


Fig. 14. Employee Request independent component.

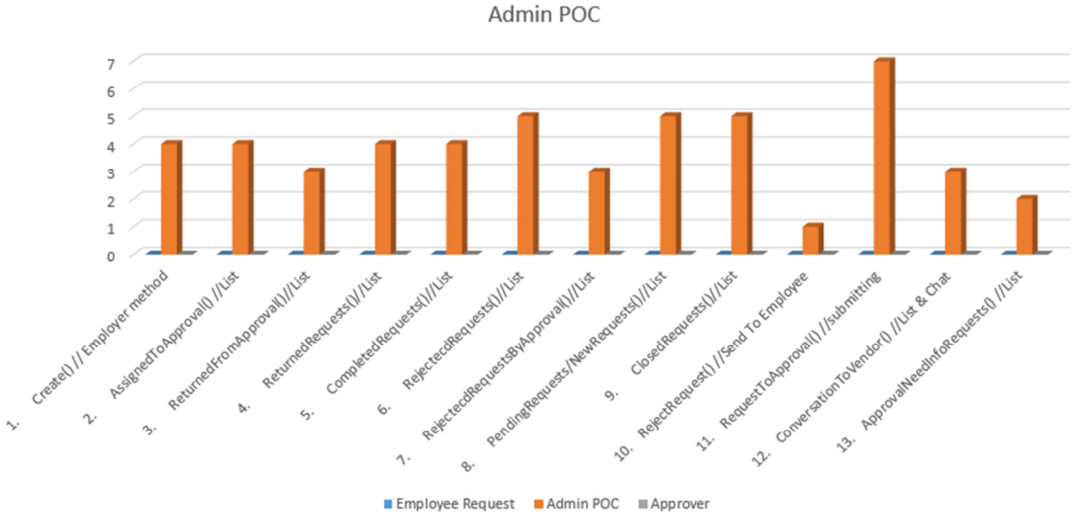


Fig. 15. Admin POC component independent from the employee Request component.

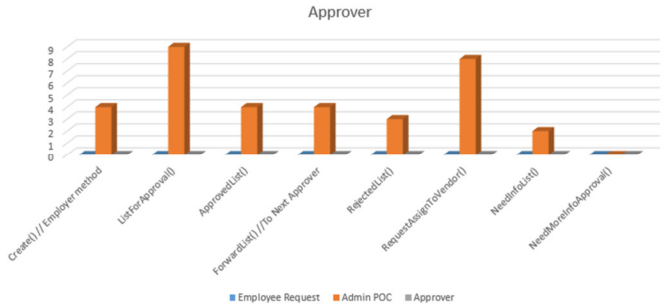


Fig. 16. Approver Component independent from the employee Request component.

Figure 14 provides a clear representation of the independent components within the system. It illustrates the components that are completely detached from the Admin POC and Approver components. This diagram effectively conveys the autonomy and independence of these components, demonstrating their ability to operate independently without relying on the Admin POC and Approver components. The visual depiction in Figure 14 emphasizes the distinct separation of these components from the rest of the legacy system, underscoring their self-sufficiency and freedom from external dependencies.

Figure 15, on the other hand, showcases an independent component that is entirely detached from the employee Requester component. This diagram illustrates the clear separation of this component from the employee Requester component, highlighting its autonomy and self-reliance. From Figure 15, it becomes evident that this independent component's methods are distinct and separate from those of the employee Requester component. This delineation emphasizes the component's ability to function independently, without any reliance on the employee Requester component.

In contrast, Figure 16 presents a component that is entirely independent of the employee Requester component, but it remains dependent on the Admin POC. This diagram depicts the component's autonomy and independence from the employee Requester component, while also highlighting its reliance on the Admin POC component.

By examining Figures 14-16, system designers and developers can gain a comprehensive understanding of the independence and interdependencies among components within the system. These visual representations offer valuable insight into the architecture and relationships between components, enabling informed decision-making for system optimization, modularity, and scalability.

## V. CONCLUSION

The current work presents a novel approach in identifying the dependent and independent components of a monolithic legacy system. This research gives awareness to understand and increase the scalability of an already developed monolithic legacy system. The favorable primary position of the proposed method is the accomplished high cohesion and low coupling of components. These useful points assume the job of tackling the issues which require software evolution. To the best of our knowledge, no published study proposes any comprehensive technique or framework to find the components with relations and no method has yet been proposed on removing the dependencies of dependent components. From the existing literature, four research questions were derived on the bases of issues regarding monolithic legacy systems. In this paper, the significant section of the proposed solution, where the operations are executed, provides proof of results in terms of validation with all the aspects of the research results. These experiments were performed on an industry-based project.

In future work, the creation of an automated tool that helps find the dependent and independent components of the project is needed based on the proposed method and for architecture recovery.

## REFERENCES

[1] C. Burstedde, J. A. Fonseca, and S. Kollet, "Enhancing speed and scalability of the ParFlow simulation code," *Computational Geosciences*, vol. 22, no. 1, pp. 347–361, Feb. 2018, https://doi.org/10.1007/s10596-017-9696-2.

[2] H. Ibrahim, B. H. Far, and A. Eberlein, "Scalability improvement in software evaluation methodologies," in *International Conference on Information Reuse & Integration*, Las Vegas, NV, USA, Aug. 2009, pp. 236–241, https://doi.org/10.1109/IRI.2009.5211557.

[3] L. Duboc, E. Letier, D. S. Rosenblum, and T. Wicks, "A Case Study in Eliciting Scalability Requirements," in *16th IEEE International Requirements Engineering Conference*, Barcelona, Spain, Sep. 2008, pp. 247–252, https://doi.org/10.1109/RE.2008.22.

[4] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issues and challenges," *Computer*, vol. 39, no. 10, pp. 36–43, Jul. 2006, https://doi.org/10.1109/MC.2006.362.

[5] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in *18th International Symposium on Computational Intelligence and Informatics*, Budapest, Hungary, Nov. 2018, pp. 149–154, https://doi.org/10.1109/CINTI.2018.8928192.

[6] D. Escobar *et al.*, "Towards the understanding and evolution of monolithic applications as microservices," in *XLII Latin American Computing Conference*, Valparaiso, Chile, Oct. 2016, pp. 1–11, https://doi.org/10.1109/CLEI.2016.7833410.

[7] A. G. Salinger *et al.*, "Albany: using component-based design to develop a flexible, generic multiphysics analysis code," *International Journal for Multiscale Computational Engineering*, vol. 14, no. 4, pp. 415–438, 2016, https://doi.org/10.1615/IntJMultCompEng.2016017040.

[8] C. J. M. Geisterfer and S. Ghosh, "Software component specification: a study in perspective of component selection and reuse," in *Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, Orlando, FL, USA, Feb. 2006, https://doi.org/10.1109/ICCBSS.2006.26.

[9] D. Liu, C.-H. Lung, and S. A. Ajila, "Adaptive Clustering Techniques for Software Components and Architecture," in *39th Annual Computer Software and Applications Conference*, Taichung, Taiwan, Jul. 2015, vol. 3, pp. 460–465, https://doi.org/10.1109/COMPSAC.2015.256.

[10] S. S. Yau, C. Taweponsomkiat, and D. Huang, "A Framework for Extensible Component Customization for Component-based Software Development," in *Sixth International Conference on Quality Software*, Beijing, China, Oct. 2006, pp. 369–376, https://doi.org/10.1109/QSIC.2006.1.

[11] S. A. K. Ghayyur, A. Razzaq, S. Ullah, and S. Ahmed, "Matrix Clustering based Migration of System Application to Microservices Architecture," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 1, pp. 284–296, Jan. 2018, https://doi.org/10.14569/IJACSA.2018.090139.

[12] W. Chengjun, "Architecture Driven Component Development for Top-Down Software Reuse," in *International Conference on Computer Science and Software Engineering*, Wuhan, China, Dec. 2008, vol. 5, pp. 1349–1352, https://doi.org/10.1109/CSSE.2008.87.

[13] J. S. Cuadrado, E. Guerra, and J. de Lara, "A Component Model for Model Transformations," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1042–1060, Aug. 2014, https://doi.org/10.1109/TSE.2014.2339852.

[14] N. Parlavantzas, M. Morel, V. Getov, F. Baude, and D. Caromel, "Performance and Scalability of a Component-Based Grid Application," in *International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, Mar. 2007, pp. 1–8, https://doi.org/10.1109/IPDPS.2007.370416.

[15] P. Geyer and S. Singaravel, "Component-based machine learning for performance prediction in building design," *Applied Energy*, vol. 228, pp. 1439–1453, Oct. 2018, https://doi.org/10.1016/j.apenergy.2018.07.011.

[16] D. Chaudhari, M. Zulkernine, and K. Weldemariam, "Towards a ranking framework for software components," in *28th Annual ACM Symposium on Applied Computing*, New York, NY, USA, Mar. 2013, pp. 495–498, https://doi.org/10.1145/2480362.2480458.

[17] J. Cubo and E. Pimentel, "DAMASCo: A Framework for the Automatic Composition of Component-Based and Service-Oriented Architectures," in *5th European Conference*, Essen, Germany, Sep. 2011, pp. 388–404, https://doi.org/10.1007/978-3-642-23798-0_41.

[18] A. L. Martinez-Ortiz, D. Lizcano, M. Ortega, L. Ruiz, and G. Lopez, "A quality model for web components," in *18th International Conference on Information Integration and Web-based Applications and Services*, Singapore, Asia, Nov. 2016, pp. 430–432, https://doi.org/10.1145/3011141.3011203.

[19] M. Vianden, H. Lichter, and A. Steffens, "Experience on a Microservice-Based Reference Architecture for Measurement Systems," in *21st Asia-Pacific Software Engineering Conference*, Jeju, Korea (South), Dec. 2014, vol. 1, pp. 183–190, https://doi.org/10.1109/APSEC.2014.37.

[20] J. Kaur and P. Tomar, "Validation of Software Component Selection Algorithms based on Clustering," *Indian Journal of Science and Technology*, vol. 9, no. 45, pp. 1–4, Dec. 2016, https://doi.org/10.17485/ijst/2016/v9i45/106369.

[21] A. Alkhalid, C.-H. Lung, D. Liu, and S. Ajila, "Software Architecture Decomposition Using Clustering Techniques," in *37th Annual Computer Software and Applications Conference*, Kyoto, Japan, Jul. 2013, pp. 806–811, https://doi.org/10.1109/COMPSAC.2013.132.

[22] G. Shahmohammadi, S. Jalili, and S. M. H. Hasheminejad, "Identification of System Software Components Using Clustering Approach," *Journal of Object Technology*, vol. 9, no. 6, pp. 77–98, 2010, https://doi.org/10.5381/jot.2010.9.6.a4.

[23] S. Marru, M. Pierce, S. Pamidighantam, and C. Wimalasena, "Apache Airavata as a Laboratory: Architecture and Case Study for Component-Based Gateway Middleware," in *1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, Portland, OR, USA, Jun. 2015, pp. 19–26, https://doi.org/10.1145/2753524.2753529.

[24] I. Hussain, A. Khanum, A. Q. Abbasi, and M. Y. Javed, "A Novel Approach for Software Architecture Recovery using Particle Swarm Optimization," *The International Arab Journal of Information Technology*, vol. 12, no. 1, pp. 32–41, 2015.

[25] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, Sep. 2011, https://doi.org/10.1109/TSE.2010.83.

[26] G. Toffetti Carughi, S. Brunner, M. Blochlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *1st International Workshop on Automated Incident Management in Cloud*, Bordeaux, France, Apr. 2015, https://doi.org/10.1145/2747470.2747474.

[27] K. K. Chahal and H. Singh, "A Metrics Based Approach to Evaluate Design of Software Components," in *International Conference on Global Software Engineering*, Bangalore, India, Aug. 2008, pp. 269–272, https://doi.org/10.1109/ICGSE.2008.29.

[28] J. Chen, W. K. Yeap, and S. D. Bruda, "A Review of Component Coupling Metrics for Component-Based Development," in *WRI World Congress on Software Engineering*, Xiamen, China, Dec. 2009, vol. 4, pp. 65–69, https://doi.org/10.1109/WCSE.2009.391.

[29] S. K. Mishra, D. S. Kushwaha, and A. K. Misra, "Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering.," *Journal of Object Technology*, vol. 8, no. 5, pp. 133–152, 2009, https://doi.org/10.5381/jot.2009.8.5.a3.

[30] A. Seriai, S. Sadou, H. Sahraoui, and S. Hamza, "Deriving Component Interfaces after a Restructuring of a Legacy System," in *IEEE/IFIP Conference on Software Architecture*, Sydney, NSW, Australia, Apr. 2014, pp. 31–40, https://doi.org/10.1109/WICSA.2014.27.

[31] S. R. Idate, T. S. Rao, and D. J. Mali, "Context-Based Aspect-Oriented Requirement Engineering Model," *Engineering, Technology & Applied Science Research*, vol. 13, no. 2, pp. 10460–10465, Apr. 2023, https://doi.org/10.48084/etasr.5699.

[32] M. O. Odhiambo and P. O. Umenne, "NET-COMPUTER: Internet Computer Architecture and its Application in E-Commerce," *Engineering, Technology & Applied Science Research*, vol. 2, no. 6, pp. 302–309, Dec. 2012, https://doi.org/10.48084/etasr.145.

[33] M. N. A. Khan, A. M. Mirza, M. Shahid, R. A. Wagan, and I. Saleem, "State of Quality Engineering Practices: The Pakistan Perspective," *Engineering, Technology & Applied Science Research*, vol. 10, no. 5, pp. 6309–6315, Oct. 2020, https://doi.org/10.48084/etasr.3782.

[34] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-Based Reliability Prediction with the Palladio Component Model," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1319–1339, Aug. 2012, https://doi.org/10.1109/TSE.2011.94.

[35] J. ZhanG, X. Ban, Q. Lv, J. Chen, and D. Wu, "A component-based method for software architecture refinement," in *29th Chinese Control Conference*, Beijing, China, Jul. 2010, pp. 4251–4256.

[36] K. Sartipi, "Software architecture recovery based on pattern matching," in *International Conference on Software Maintenance*, Amsterdam, Netherlands, Sep. 2003, pp. 293–296, https://doi.org/10.1109/ICSM.2003.1235434.

[37] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of Component-Based Architecture from Object-Oriented Systems," in *Seventh Working IEEE/IFIP Conference on Software Architecture*, Vancouver, BC, Canada, Feb. 2008, pp. 285–288, https://doi.org/10.1109/WICSA.2008.44.

[38] K.-K. Lau and S. Di Cola, "(Reference) architecture = components + composition (+ variation points)?," in *1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures*, Montreal, QC, Canada, Dec. 2015, pp. 1–4.

[39] D. E. Bernholdt *et al.*, "A Component Architecture for High-Performance Scientific Computing," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, May 2006, https://doi.org/10.1177/1094342006064488.

[40] H. Algestam, M. Offesson, and L. Lundberg, "Using components to increase maintainability in a large telecommunication system," in *Ninth Asia-Pacific Software Engineering Conference*, Gold Coast, QLD, Australia, Dec. 2002, pp. 65–73, https://doi.org/10.1109/APSEC.2002.1182976.

[41] P. Rana and R. Singh, "A soft computing approach to optimize component based software complexity metrics," *Journal of Theoretical and Applied Information Technology*, vol. 97, pp. 1200–1212, Feb. 2019.

[42] D. P. Lupp, M. Hodkiewicz, and M. G. Skjaeveland, "Template Libraries for Industrial Asset Maintenance: A Methodology for Scalable and Maintainable Ontologies," *CEUR Workshop Proceedings*, vol. 2757, pp. 49–64, 2020.