Journal of Theoretical and Applied Information Technology
15st June 2026. Vol.104. No.11

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AUTOMATED DOMAIN ANALYSIS FOR SOFTWARE REUSE USING PACKAGE ABSTRACTIONS AND GENERICS

Dr B. JALENDER [1] , Dr. T. VENKATA RAMANA [2] , BANOTH SAMYA [3], Dr L. KIRAN KUMAR REDDY [4] , Dr. KACHAPURAM BASAVARAJU [5] . GUGULOTHU VENKANNA [6].

[1]Associate Professor, Department of AIML and IoT, VNR Vignana Jyothi Institute of Engineering & Technology, Hyderabad, India.
[2] Associate Professor, Department of CSE – AIML, CVR COLLEGE OF ENGINEERING, Hyderabad, India.
[3] Associate Professor, Department of CSE, CVR COLLEGE OF ENGINEERING, Hyderabad, India.
[4]Associate Professor, Department of CSE - Data Science, Geethanjali College of Engineering and Technology, Cheeryal, Hyderabad, India
[5] Associate Professor, Department of AI , Anurag University, Hyderabad, India
[6] Associate Professor, CSE Department, Sreenidhi Institute of Science and Technology, Hyderabad, India
E-mail: [1]jalender_b@vnrvjiet.in,[2]meetramana1204@gmail.com, ,[3]samyabanoth@gmail.com, [4]kirankumarlakkadi@gmail.com, [5]kbrajuai@anurag.edu.in, [6]venkanna.g@sreenidhi.edu.in

ABSTRACT

Domain analysis aims to identify and design reusable components for families of products, defining the necessary domain roles, processes, models, and architectures. While existing literature provides guidelines and techniques for domain analysis, our work takes a more practical approach by automating these principles. We implemented a system designed to address the challenges of Design for Reuse (DFR) in detail. We have developed reusability guidelines, which include that a good package can be used to help facilitate a good reuse experience, similar to what Python has developed for its usage. The concept of package as a strong reuse mechanism can be accomplished using Ada methods and designs through proper usage of private types, separating specification from implementation, and using generics for the parameterization of the users package. Each of these methods can be used to produce a viable reusability product regardless of the language being used for the creation of the product. In this instance of this methodology, all composite types have been categorized into abstract data structures.
Keywords: *Software reuse, Domain Analysis, reusability, Data Structures, components, CBSE.*

1. INTRODUCTION

Software Reuse is one of the key element in any industry that that drive business efficiency. The idea of reuse has long history and it is extensively used for IT systems and software development. The emergence of Business Process Management (BPM) practices and the interest in managing processes are leading to the growing importance and aspiration for reusing processes [1]. In general, reusable components tend to be much more stable than non-reusable components To achieve the goal of production of reusable components, the challenge of what makes a component more reusable has to be resolved. Therefore, we adopted a more pragmatic way of dealing with this challenge by automating reuse guidelines for domain abstractions and attributes for reuse (Domain analysis for reuse) as well as the identification of features of a programming language that influence component reusability (Language analysis for reuse). For instance, some languages (C, C++, Java, Python) provide support for reuse explicitly. Our method tries to unify guidelines on domain analysis and language features [2].

Journal of Theoretical and Applied Information Technology
15st June 2026. Vol.104. No.11

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

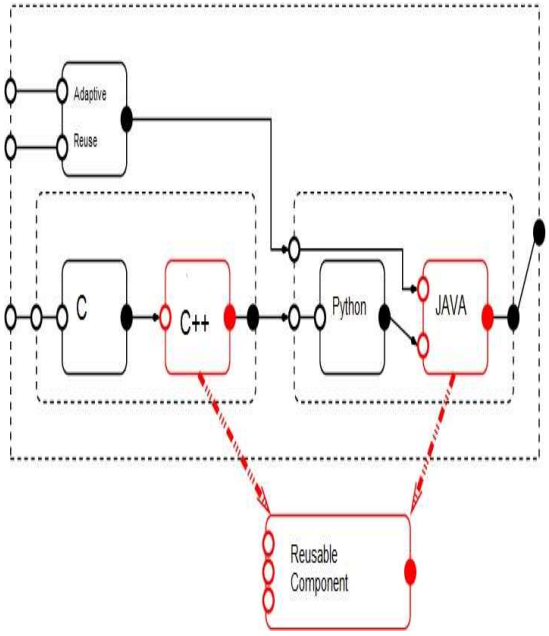

Figure 1. Adaptive Reuse

The results of this analysis indicate that, although the reusable components demonstrated a lower defect density than the new components, it is the inverse when measuring defect levels by severity of defects of the component. This would indicate that, although there were many defects associated with the reusable components after delivery, the total number of defects associated with the reusable components would be greater than that for the new components but would be less for low severity defects associated with the reusable components than for the new components. According to previous research conducted by other researchers, reusable components will change more frequently than new components because reusable components must be compliant with multiple different systems and different domains. In general, reusable components tend to be much more stable than non-reusable components [3].

Domain analysis aims to identify and design reusable components for families of products, defining the necessary domain roles, processes, models, and architectures. While existing literature provides guidelines and techniques for domain analysis, our work takes a more practical approach by automating these principles.

We implemented a new system to address the challenges of Design for Reuse (DFR) in detail. Using existing research as a foundation, we formulated specific reuse guidelines and automated them to facilitate the development of reusable software components. Our process defines DFR through several key stages, identifying domain abstractions and classification (domain-oriented reuse), language-oriented reuse, reuse assessment, and reuse improvement.

We have recently undertaken the task of applying the Guidelines developed through this research and experience toward creating reusable, scalable architectures for large-scale industrial implementations [4]. Using this approach will help other organizations identify reuse options earlier in the development cycle through the use of reuse assessment tools. The methodology been found to provide the following advantages in this area:

1. The automated identification of reusable architectural abstractions, attributes, and architectures based on domain classifications.

2. The automation of providing reuse guidelines to assist developers in constructing reusable items.

3. Support of software engineers in the reuse assessment and improvement process.

4. The ability to create reusable components based on automated template modelling.

5. The ability to generate reusable components with a high likelihood of reuse in future implementations.

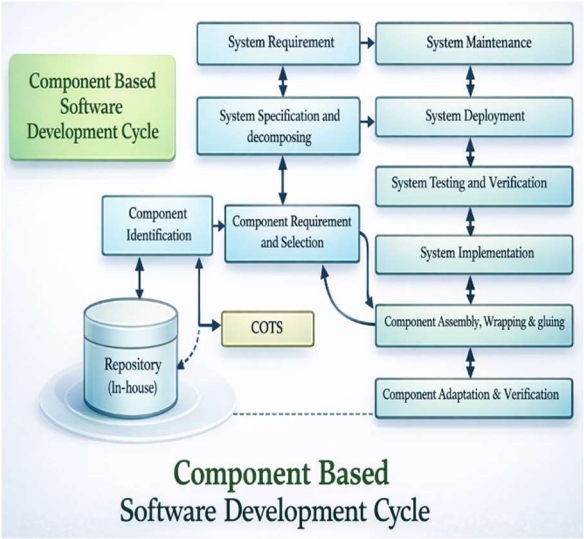

Figure 2 .Component Based Software Development Cycle

Domain analysis aims to identify and design reusable components for families of products, defining the necessary domain roles, processes, models, and architectures. While existing literature provides guidelines and techniques for domain

analysis, our work takes a more practical approach by automating these principles.

Component-based software engineering (CBSE) provides a systematic approach to managing software complexity by combining the modular, reusable, and independently deployed components. CBSE focuses on building large-scale systems by assembling existing, well-defined software components, rather than developing general-purpose solutions from scratch. These principles are widely used in software development life cycle to improve maintainability, reduce development time, and improve system reliability [3].

Most of the latest network paradigms essentially embody the principles of Component-based software engineering, even if they are not clearly defined. The as software-defined networking architecture is based on modular control plane reusable components. The network function virtualization structure decomposes network services into reusable virtual network functions.

The cloud-based proprietary network adopts a microservice-based design, emphasizing loose coupling and independent expansion. Similarly, the architecture based on 5g services realizes network functions as modular services with standardized interfaces, thereby realizing dynamic network combination and slicing. These changes show that CBSE is not only suitable for the web, but is actually the main enabler of modern network architecture [4].

Existing research tends to analyze the SDN, NFV, cloud, and 6G technologies in isolation, usually focusing on specific protocols, performance optimization, or implementation details. Of course, there is not enough research to systematically explore these examples through the reuse of software components. Therefore, so far, not enough attention has been paid to the broader architectural impact of CBSE on network performance, extensibility, and long-term development [5].

In addition, as the 5g network becomes more dynamic and distributed. The network performance of 6g is going to exceed the original bandwidth and latency. Understanding how CBSE-based design affects these factors is essential for the development of new 6g reliable and future-oriented network systems[6].

Based on these observations, this Research paper discusses the relationship between the reuse of software components and the latest trends in computer networks, and positions component-based software engineering as a unified architectural paradigm [7]. The study examined modern network technology centred on CBSE, examined the impact of its performance and extensibility, and identified key research challenges and future directions. Bridging the gap between software development and network research, this work aims to promote the development of modular, scalable and high-performance network systems.

2. PROBLEM STATEMENT AND CONTRIBUTIONS

Software-based network technology is widely used for the systematic role of component-based software engineering (CBSE) in shaping modern network architecture. The technology has not yet been fully understood by the end user. Existing research usually studies SDN, NFV, cloud networks, and 5G networks [8]. This technology will focusing on protocol development, virtualization technologies, or management mechanisms. Therefore, there is a lack of a unified architectural perspective to explain how software component reuse, composability, and modular design principles together affect network performance, extensibility, and evolution.

In addition, traditional network design methods are rely on close collaboration and vendor-specific deployment, limiting reuse and hindering rapid innovation. Even in modern programmable networks, the lack of standardized component models and combination strategies that take into account performance can lead to inefficient use of resources, increasing latency and work complexity. These problems are further exacerbated in large-scale and heterogeneous environments such as multi-cloud infrastructure and 5g-enabled edge networks [9].

Hot-spots and meta-patterns provide the possibility for transforming the complex problem of porting in the simpler problem of providing to the framework new modules, specific for the new target environment. Instead of simply counting the number of external dependencies as a qualitative measure of the portability of a framework, a better strategy should so take into account the adaptation mechanisms built in the framework in order to deal with portability problems.

The adaptability of the framework depends from two different specific criteria: generality and modularity. Generality is defined, for an object-oriented framework, as the capability of meeting the requirements of different applications in which the reusable component could be instantiated. This is translated into the qualitative evaluation of the

domain coverage supplied by the possibility of adaptation through the provided hot-spots.

A judgement on this point is useful during design with reuse especially if made upon the hot-spots providing the needed domain-specific adaptation mechanisms. In other words, a reuser should not have to worry about the modularity of the entire structure, since hot-spots should be sufficient to customize the framework to new needs. What is interesting is instead the modularity of the provided adaptation points. In this respect, our view counters the normal approaches in which modularity is measured on the global structure.

Therefore, the main problem discussed in this article is the lack of a comprehensive CBSE-oriented framework and analytical perspective to systematically link the software reuse of software components with the latest developments in computer networks, while clearly considering performance and performance [10].

3. COMPONENT-BASED SOFTWARE ENGINEERING FUNDAMENTALS

Component-based software engineering focuses on building systems using reusable components, self-contained software reusable components with well-defined interfaces. The core CBSE principles include modularity, loose coupling, and high cohesion. In a distributed environment, CBSE supports independent deployment, extensibility, and fault isolation, especially suitable for modern network systems [11].

3.1 Related work

Early research on component-based systems emphasized modularity and reuse in software-intensive applications. Szyperski established a basic concept for component software, highlighting the combination of independent deployment and interface-based [12].

In the field of networking, the study of software-defined networks explores programmable control planes, but usually focuses on controller design rather than component reuse methods. NFV research has proven the benefits of virtualization, but there is a lack of a formal CBSE framework for VNF combinations [13].

Recent work on cloud-native networks and 5g service-based architectures implicitly adopts the idea of CBSE, however, the unified CBSE-centered analysis across the network paradigm is still limited. Evolution to a software reuse network are shown in below figure 3 and 4.This article addresses this gap by providing an overall perspective based on CBSE.

If software requirements are being thought of from a generic perspective in order to serve a wider audience, they will be more abstract and disconnected from the direct user requirements. That means the drive for developing is not coming from the core business it is something which has to be… invented. Next level (lower) would be frameworks, then services, libraries, components and then code. The cost involved would go up as the generality increases.

Building new languages, database engines, frameworks involve a huge cost. Sometimes the business might want to build engines or invent things to be ahead of the competition and maybe forecast returns in future, for that the other side of business needs to sustain the cost of building reusability. Perhaps, an analysis should be done for why there are similar requirements in different software's. Sometimes there might be business reasons for having similar software's.

3.2. Evolution to a software-centric network

The decoupling of the control and data layers, network virtualization, and API-driven management transform the network into a programmable platform. These developments have enabled CBSE practices to be directly applied, allowing network functions to be designed as reusable software components [14].

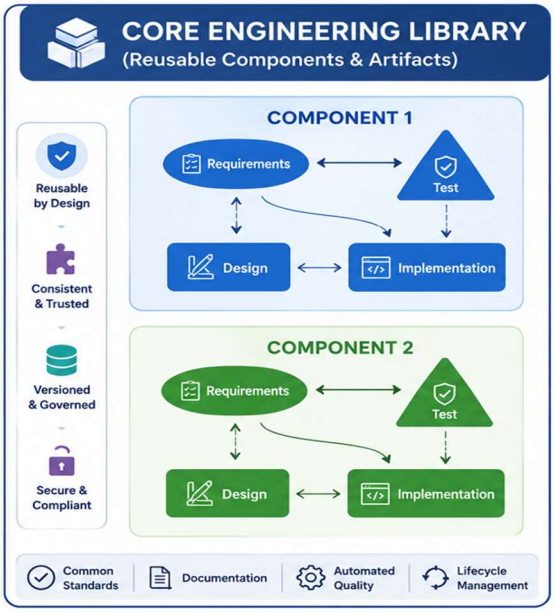

Figure 3. Evolution to a software reuse network

Journal of Theoretical and Applied Information Technology
15st June 2026. Vol.104. No.11

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

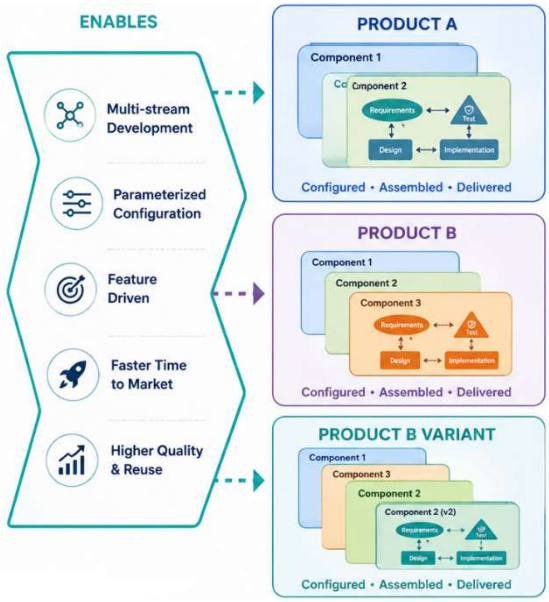

Figure 4. Evolution to a software reuse centric network

4. THE ROLE OF SOFTWARE COMPONENTS

Software components play a vital role in the industrial development process. Various artifacts are generated throughout the software lifecycle, each holding significance for the final product. Reusing these work products offers clear benefits: reduced costs, lower maintenance requirements, and decreased testing efforts. However, as components evolve in different environments within a repository, designing consistently reusable components remains a critical issue.

While various approaches exist in the literature, many suffer from significant limitations, often treating code as components in isolated environments. Furthermore, the incompatibility of legacy components presents a major challenge. This motivates our proposed approach: generating reusable design components from legacy systems to enable reuse across different environments, thereby facilitating faster software development[7].

4.1 Objectives and Results

The focus of the current study is to determine how to create an environment conducive to reuse, and to determine if it is feasible to automatically evaluate and manipulate the reuse properties of a software component through automation. Our research specifically addresses the following issues:

1. The identification of domain abstractions and the classification of those abstractions (domain-oriented reuse).

2. The ability to determine the potential for reuse of a software component at an early stage of development.

3. To make the component more reusable via automation.

The system has been shown to be capable of automatically identifying, evaluating and enhancing the reusability of a software component using information derived from both domain knowledge and programming language knowledge. Our work goes beyond mere evaluation of a component's reusability, and has demonstrated that a sizable number of large industrial applications can be reused by using generic component templates and generic architectures.

Key Improvements Made:

1. **Clarified the "Ada vs. Python" comparison:** The original text was a bit confusing about whether it was using Python or Ada. The rewrite clarifies that while Python's packaging is the concept, Ada's specific features (generics/private typing) enable specific guidelines, but the system is language-agnostic.
2. **Standardized Terminology:** Changed "work products achievement" to "final product" and "dire need of designing... is still an issue" to "designing... remains a critical issue" for better readability.
3. **Structured the Objectives:** The goals were buried in the text. I separated them into clear objectives (Identify, Assess, Improve) to make the contribution of the paper obvious.
4. **Legacy Components:** I smoothed the transition into the discussion of legacy components, explaining why they are a problem (incompatibility) and how this solves it.

5. FRAMEWORKS AND PATTERNS

The software engineering concept that is most frequently associated with frameworks is a pattern. Object-oriented design patterns are templates for describing a possible design experience in terms of a solution to a general problem which occurs repeatedly [9]. The explanations in description of design patterns almost always include an information on a problem which pattern covers, on

the context within which this pattern solves that problem, on trade-offs and forces influencing design of the hook structure that pattern represents, i.e. an examples for applying such a kinda solutions.

Unlike frameworks, they are not a full-blown system themselves: they just solve one particular kind of design question. In addition, they are independent from any specific programming language as these only represent design experience in a way that preserves the concept of a respective design subset[10].

Patterns are found in the majority of well-designed software. Frameworks are no exceptions in this regard: design patterns can also be used for constructing the framework's architecture and devising adaptation mechanisms in its hot-spots. Specifically the reoccurring patterns detected in hot-spots are generally categorizable into 7 combinations known as meta-patterns[11]. Meta-patterns are light-weight; they have a simple shape and are composed of template methods written around small hook methods.

Hooks parametrize the behaviour of the template, and by overriding these hooks (through sub classing) a reuser can easily adapt the template to altered requirement. A complete description of the purpose, structure and applications of meta-patterns is given in [12].

At a higher degree of generality, meta-patterns are relevant as they define the general "scheme" hidden behind a given adaptation mechanism located at framework-hot spot. for a software engineer who knows the pattern space organization of an arbitrary meta-pattern, it becomes easier to understand its instantiation in each framework and therefore how the hot-spot is adapted. This latter are general, fundamental adaptation ontology (with semantic characterization) abstract in the relevant aspects of various adaptation mechanisms[13].

5.1 Existing reusability assessment models

Sadly, framework production demands a very significant amount of work when contrasted to regular ad-hoc development. In most cases, costs are much higher than the cost of developing a single application. In this perspective frameworks are long term investments that are justified only if they are used in multiple applications dealing with the particular application domain. Furthermore, the framework technology is not mature: some of the issues related to framework compatibility, language binding and poor tool support are still open.

An other essential question, which has not yet been completely addressed is the optimal selection of the hot-spots structure that the framework expose to its reusers. Only framework reuse can offer large-grain domain-specific components that are the size of an architectural subsystem and that can be composed to create complex application systems.

In this sense the reusability Metric is the cornerstone.An evaluation of the reusability of a framework is an essential beginning to any economic analysis of corporate reuse strategy. It underlies analysis of what is happening activity wise and where remedial or improvement action can be taken. Without such a reusability analysis, only a retrospective comparison with development in reuse (e.g. via productivity data) may be possible[14].

Not having basic knowledge, it is impossible to formally treat the alternatives of with reuse and from scratch at least in the initial stage. Also the framework engineering team cannot figure out what would be a set of evolution directions which give to the collection of components more chances to reimplemented Based on our examination of the specific aspects of oops, we investigated past approaches to software reusability. What did unfold is that the current proposals for measuring software reusability are not adequate to our scenario, due them express characteristics which are mostly suitable to single component reuse.

. The main problems are:

- Too broad a concern of the complete structure instead of on the parts where software component adapts. The majority of reusability models are conceived for general reusable components, articulating the idea that when designing with reuse one must know and possibly modify the interfacing of developing concerns at a global level.
- Reusing software in a way that allows other developers to understand the code is not an efficient use of resources. Frameworks, on the other hand, provide points where modifications to existing code can be made at minimal cost. Therefore, in the previous section, we determined that the existing methods for determining reusability do not measure all of the reusability of software since there are also other elements involved in determining the value of some software packages that are reused. For example, the method proposed by Prieto-Diaz and Freeman uses a broad classification of software packages [8] and

only takes into account the total program size when assessing the reusability. The NATO standard is another example that provides guidance on reusing software by assessing only the total cyclomatic complexity of a software project [15].

- Metrics applied to code rather than design; i.e., the design of large software frameworks (or subsystems) is far more telling than the code itself. The design houses a lot of information that gets lost or concealed in the implementation details found in the code. The foundation of providing extensibility needs to be addressed during the development of the design, not afterwards during the coding phase. At the design stage is the appropriate time to assess the reusability of the software asset, which enables the developer to properly organise and structure the software project prior to its start. Early stage planning for future reuse leads to cost savings all around. Many of the metrics involved in the reusability assessment of software assets, such as size (lines of code) etc. are not applicable to the design stage due to the fact that code does not exist at this point. Other functional measurement metrics (function points etc.) are available for use at the design stage. The introduction of the Unified Modelling Language (UML)[1], has provided a standard - de-facto, therefore it is now possible to furnish a general description of design metrics based upon an accepted glossary and vocabulary[16].

• There is a low priority placed on the idea of using design patterns in their assessment of reusability. Unlike other models that assess the overall reusability of a framework, all aspects relating to a framework's design, including hot-spots, templates and hooks will be taken into account when creating the metrics or qualitative value for a framework's assessment of reusability. Also, since many design patterns are used to define the architecture of a framework, the assessment of reusability should be able to factor in the documentation for those design patterns. To date, none of the reusability models consider the value of documenting design patterns. The absence of attention to this aspect of reusability has a number of negative consequences. For example, the resulting designs for most hot-spots will vary greatly based on the combination of template and hook characteristics found within that hot-spot; however, because of the way templates, hot-spots and hooks connect with other framework classes, the impact that those combinations have on the degree of coupling a reuser has during the process of adapting a framework are unaccounted for. In addition, while careful use of design patterns during the design of frameworks may greatly reduce the amount of time or resources required by a reuser to adapt to the new asset, this aspect does not receive any attribution during the evaluation [17].

6. FACTORS AFFECTING FRAMEWORK REUSABILITY

The main factors which affect the reusability of a software component can be established through an examination of the activities performed during reuse. These activities include:

- Porting the component into a new environment. The primary consideration for porting is the degree of ease with which a software component can be moved from its original environment(s) to new ones. This characteristic relates directly to how much independence the reusable component has from its immediate environment, e.g., the operating system, libraries, and other bases used by the environment(s) where it was originally created. It should also be noted that within the context of the specialization of the reusability model for software development, porting was removed as a major consideration. This would probably be due to the fact that the design of a software product can generally be implemented using many different programming languages and supporting tools using lower-level programming languages. However, we believe that when creating a design for a software.[18]

- Allowing the component to adapt to a different functional requirement of a New System requires a high degree of adaptability to the new requirements. Adaptability in this context is defined as the ease with which a component may be modified to achieve a goal requiring different functionality than that originally intended. The principal purpose of framework development is to help reduce the costs associated with integrating a subsystem into a New System with a slightly different functional requirement than the original. Therefore, from our perspective, Adaptability represents the single largest advantage of frameworks over all other types of Reusable Software's in regard to adaptability [19].

- Recognizing how the Component functions to determine whether it meets the New Functional Requirements, and to modify it for those requirements, requires a high degree of understandability of the component's function. Therefore, the degree of Effort required by a Developer to determine the logical concept on which a Software Asset is constructed, and to determine how the asset could be employed, has a direct correlation to the understandability of the asset. Understandability is more a function of documentation than good software design. The only components of a Framework Design that are impacted by this factor are the Hot Spots; the basis for the Adaptation Points must be clearly defined and documented extensively, along with providing numerous examples, scenarios, and illustrations, and the complexity associated with it in relation to other components must be kept to a minimum [20].

To make sure that the reused component does not introduce any additional system risks, we must be able to determine that it has a high probability of performing its intended function properly with no failure for a specified period when implemented in an environment different than the one from where it was originally developed and/or certified. Typically, it is not possible to assess the impact of this factor on the usability of a framework until after multiple instances of that same framework have been developed and tested [21][22].

7.CONCLUSION AND FUTURE WORK.

For the purposes of this article, we have defined a number of specific criteria for assessing the reusability of a framework. These criteria allow for a discussion that brings the evaluation of a framework closer to the work of software engineers—connecting the design and implementation of frameworks with more general factors defined at the executive project management level. In addition, the criteria also allow for an easier way to determine how reusable a framework is because the criteria are more directly related to design, implementation, and the actual source code structure of a framework. Furthermore, using the criteria during the design and implementation phases will allow developers to modify their designs and implementations in a manner that improves the reusability of their products. The purpose of this paragraph is to create and present a small number of specific criteria for each of the identified high-level factors associated with framework reusability. Each of the criteria presented in this paragraph takes advantage of the specific features of object-oriented frameworks as a basis for defining the assessment of reusability of a framework.

Portability is one of the most important considerations and is associated with the amount of the framework's dependence on functional subsystems (including the operating systems and libraries) that are externally available for access. Typically, for frameworks to be portable, they should have as few dependencies on the surrounding environment as possible. The technical dependencies include things like system calls for threading or process management, accessing databases, accessing object-oriented data stores, and loading or using external libraries, as well as some non-technical elements, such as a specific look and feel for the user interface or a specific style of programming, which could also be a cause of limited portability.

Future work includes AI-driven component orchestration, formal verification of network components, and network architecture based on adaptive CBSE. These directions have opened up new opportunities for doctoral research. Although this article establishes the conceptual and architectural link between component-based software engineering (CBSE) and the latest developments in computer networks, several promising research directions are still open and deserve further research.

First of all, the quantitative performance evaluation of CBSE-based network architecture is an important research direction in the future. Large-scale experimental verification can be carried out using network simulators (such as Mininet, NS-3) and real test platforms to analyze indicators such as latency, throughput, control plane overhead, and failure recovery time. A comparative study between monolithic network functions and component-based virtualized functions will provide empirical evidence of scalability and efficiency improvement.

The new Research has determined that a significant amount of the software composition challenge is the result of architectural mismatches, meaning that the dependencies of each subsystem concerning the overall structure or behaviours of the environment are incorrectly assumed. For example, this can happen because the subsystem has certain requirements that are based upon the presumed conditions of the environment and connection between the components or the time constraints

imposed upon the components and the architecture of the system and the way that the system was developed. Framework developers can approach the portability problem like any other adaptation problem.

8.REFERENCES:

[1] Szyperski, C., (2002) "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, Vol. 1, No. 1, pp. 1–25

[2] Heineman, G. T. & Councill, W. T., (2001) "Component-Based Software Engineering: Putting the Pieces Together", IEEE Computer Society Press, Vol. 5, No. 2, pp. 45–60

[3] Kreger, H., (2003) "Fulfilling the Web services promise", IBM Systems Journal, Vol. 42, No. 2, pp.155–166

[4] McKeown, N., Anderson, T., Balakrishnan, H., et al., (2008) "OpenFlow: Enabling Innovation in Campus Networks", ACM SIGCOMM Computer Communication Review, Vol. 38, No. 2, pp.69–74

[5] Kreutz, D., Ramos, F. M. V., Verissimo, P., et al., (2015) "Software-Defined Networking: A Comprehensive Survey", Proceedings of the IEEE, Vol. 103, No. 1, pp. 14–76

[6] Mijumbi, R., Serrat, J., Gorricho, J. L., et al., (2016) "Network Function Virtualization: State-of-the-Art and Research Challenges", IEEE Communications Surveys & Tutorials, Vol. 18, No. 1, pp.236–262

[7] Han, B., Gopalakrishnan, V., Ji, L., & Lee, S., (2015) "Network Function Virtualization: Challengesand Opportunities for Innovations", IEEE Communications Magazine, Vol. 53, No. 2, pp. 90–97

[8] Pahl, C., (2015) "Containerization and the PaaS Cloud", IEEE Cloud Computing, Vol. 2, No. 3, pp.24–31

[9] Dragoni, N., Lanese, I., Larsen, S. T., et al., (2017) "Microservices: Yesterday, Today, and Tomorrow", Present and Ulterior Software Engineering, Vol. 3, No. 1, pp. 195–216

[10] ETSI, (2014) "Network Functions Virtualisation (NFV): Architectural Framework", ETSI GS NFV002, Vol. 1, No. 1, pp. 1–45

[11] Foukas, X., Patounas, G., Elmokashfi, A., & Marina, M. K., (2017) "Network Slicing in 5G: Surveyand Challenges", IEEE Communications Magazine, Vol. 55, No. 5, pp. 94–100

[12] Zhang, H., Liu, N., Chu, X., et al., (2019) "Network Slicing Based 5G and Future Mobile Networks:Mobility, Resource Management, and Challenges", IEEE Communications Magazine, Vol. 55,No. 8, pp. 138–145

[13] Bonomi, F., Milito, R., Zhu, J., & Addepalli, S., (2012) "Fog Computing and Its Role in the Internet of Things", ACM MCC Workshop, Vol. 1, No. 1, pp. 13–16

[14] Medvidovic, N. & Taylor, R. N., (2000) "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, Vol. 26, No.1, pp. 70–93

[15] Bass, L., Clements, P., & Kazman, R., (2013) "Software Architecture in Practice", Addison-Wesley,Vol. 3, No. 1, pp. 85–110

[16] Buyya, R., Calheiros, R. N., & Goscinski, A., (2011) "Cloud Computing: Principles and Paradigms",Wiley Press, Vol. 1, No. 1, pp. 1–40

[17] Mao, Y., You, C., Zhang, J., et al., (2017) "A Survey on Mobile Edge Computing: The Communication Perspective", IEEE Communications Surveys & Tutorials, Vol. 19, No. 4, pp.2322–2358

[18] Villamizar, M., Garcés, O., Castro, H., et al., (2016) "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications", IEEE Computing Conference,Vol. 1, No. 1, pp. 583–590

[19] Bernstein, D., (2014) "Containers and Cloud: From LXC to Docker to Kubernetes", IEEE Cloud Computing, Vol. 1, No. 3, pp. 81–84

[20] Jamshidi, P., Pahl, C., Mendonça, N. C., et al., (2018) "Microservices: The Journey So Far and Challenges Ahead", IEEE Software, Vol. 35, No. 3, pp. 24–35

[21] S. Javanmardi, A. Nascita, A. Pescapè, G. Merlino, M. Scarpa An integration perspective of security,privacy, and resource efficiency in IoT-Fog networks: a comprehensive survey Computer. Network. (2025)

[22] R. Kong, Y. Li, W. Wang, L. Kong, Y. Liu Serving moe models on resource-constrained edge devices via dynamic expert swapping IEEE Trans. Computers, 74 (8) (2025), pp. 2799-2811.