Towards a practical framework for evaluating software architecture from a Quality perspective

Ritu Kapur¹, Sumit Kalra², Kamlesh Tiwari³ and Geetika Arora³

¹Department of Computer Science and Engineering, Indian Institute of Technology Ropar, Punjab, India.
²Department of Computer Science and Engineering, Indian Institute of Technology Jodhpur, Rajasthan, India.
³Department of Computer Science and Engineering, Birla Institute of Technology and Science, Pilani, Rajasthan, India.

dev.ritu.kapur@gmail.com, sumitk@iitj.ac.in, {kamlesh.tiwari, p2016406}@pilani.bits-pilani.ac.in

Keywords: Architectural evaluation, Pattern, Tactics, Views, Quality attributes

Abstract: A system architect uses quality requirements to design the architecture of a system. At the same time, it is also essential to ensure that the system’s final design can achieve expected quality requirements. There are several architectural evaluation frameworks, but they require a set of skills for practical usage. This work proposes a framework that enables novice software architects to infer the system’s quality requirements and tactics, given its software architectural block-line diagram. The framework does not assume any specific type of architectural diagrams. It takes an image as input, extracts various components and connections among them, and maps them to viable architectural patterns, followed by identifying the system’s corresponding quality attributes (QAs) and tactics. The framework includes specifically customized image processing techniques, semantic similarity methods, and rule-based inference techniques to assist software architects in evaluating a given design by a) evaluating an input architectural design based on the architectural patterns present in it, b) lists out the strengths and weaknesses of the design in terms of QAs, c) recommends the necessary architectural tactics that can be embedded in the design to achieve the lacking QAs. To train our framework, we developed a dataset of 2002 architectural images from nine architectural patterns such as, Client-Server, Microservices, Model View Controller, Peer-to-Peer, Broker, Publisher-Subscriber, Layered, etc. The framework achieves a Correct Recognition Rate (CRR) of 62.65% in identifying the architectural patterns with nine classes. However, the CRR becomes better with fewer classes and reaches 98.46% when architectural images of the two classes are considered. Our framework was evaluated using controlled and experimental groups using various parameters, such as accuracy, time performance, explainability, and robustness. The experimental group performed better than 150% compared to the controlled group for architectural evaluation on all the parameters.

1 INTRODUCTION

Software development generally begins with the specification of the functional and non-functional requirements of the software. The functional requirements model the functional correctness of a software, while the non-functional requirements capture the degree to which a software achieves the intended functionality. Both the functional and non-functional requirements are listed in the requirements specification of software. The inter-dependency of the functional and non-functional requirements make both critical for software quality. The non-functional requirements are specified as quality attributes (QAs), such as performance, security, and reliability.

QAs have been termed as reusable architectural building blocks used in developing an application architecture [Kim et al., 2009]. However, in order to embed the QAs in software design, one implements various software tactics. For instance, authentication tactic is incorporated into a software design to attain security; queues are incorporated to attain better performance, and so on. The design choices made by software developers among various architectural patterns and their tactics determine the software quality. Therefore, it becomes essential to evaluate the software architecture by analyzing the architectural patterns and tactics it comprises and the QAs it fulfills.
1.1 Motivation

Stated broadly, the prime objective of our work is to develop a framework that:

1. given an input architectural design image, determines and lists the QAs it meets or lacks, and
2. recommends the relevant software tactics to achieve the lacking QAs.

For instance, the pipe and filter architectural pattern distributes a task’s processing into various independent sub-tasks and is thus primarily used to attain performance, availability, and reliability. Timely detection of software quality by analyzing its architectural design can help lower the overall defect-fixing cost [Kapur and Sodhi, 2020]. Some of the critical questions in this context could be:

1. What are the significant QAs and tactics associated with architectural design?
2. How to evaluate the architectural designs using QAs?
3. What are the necessary architectural tactics that can be implemented to achieve various QAs?
4. Is there any automated method to detect the standard architectural patterns present in an architectural design?
5. Given an input architectural design image, is it possible to detect the QAs it captures or lacks in, and the necessary tactics that can be implemented to achieve the lacking QAs?

In this paper, we address these questions via an in-depth examination of QAs associated with various architectural design patterns and the necessary tactics to implement them.

1.2 Existing software architectural evaluation methods

A broad categorization of the existing evaluation methods [Abowd et al., 1997, Dobrica and Niemela, 2002] is as follows:

1. **Questioning methods** comprise asking qualitative questions on an architecture derived from the software quality aspect. These can be used to evaluate the quality of any given architecture and are classified as follows:

   (a) **Scenario:** It is a sequence of steps involving the use or modification of the system. It provides a means to characterize how well a particular architecture responds to the demands placed on it by those scenarios.

   (b) **Questionnaire:** It is a list of general and relatively open questions applicable to all architectures. The questions generally cover various topics, such as the architectural generation method or the architectural description details.

   (c) **Checklist:** It is a detailed list of questions developed after evaluating a domain-specific set of systems.

2. **Measuring methods** comprise quantitative measurements made on the architecture by addressing specific QAs, and have the following salient types:

   (a) **Experience-based methods:** These are based on the experiential knowledge of the experts [Bosch, 2000].

   (b) **Metrics-based methods:** Metrics are termed as the quantitative interpretations of some observable measurements on architectural elements, such as the fan-in/ fan-out of various components. The metrics-based evaluations focus on the metrics values extracted from various architectural components while considering the assumptions involved in such metrics [Selby and Reimer, 1995].

   (c) **Mathematical modelling-based methods:** These comprise mathematical proofs and methods for evaluating the operational QAs, such as the performance and reliability of architectural components [Reussner et al., 2003].

   (d) **Simulations, prototypes, and experiment-based methods:** Simulations, prototypes, and experimental results are often a part of the software development process and play an important role in answering various questions during software reviews. For instance, simulation results can be used to validate specific assertions. However, this tends to be an expensive approach if a prototype is specifically developed to perform the evaluation.

1.3 Basic tenets behind our system

The central idea underlying our approach (as illustrated in Fig. 1) can be stated as follows:
1. Every software has an architectural design, which comprises of several patterns.

2. Every architectural pattern incorporates several software tactics. For instance, the pipe-and-filter pattern incorporates fault detection, recovery, and performance tactics by various design concepts, such as ping (or echo), active redundancy, shadowing, spare, and so on.

3. Successful execution of software tactics leads to the fulfillment of QAs. For instance, the successful execution of recovery and performance tactics leads to the availability, reliability, and maintainability of QAs.

4. Software quality is defined as the degree to which a system, component, or process meets the specified requirements [Committee et al., 1990].

5. Software QAs can be used as metrics to measure software quality.

2 RELATED WORK

The scenario-based Architectural Analysis method (SAAM) [Kazman et al., 1994] was introduced in 1993 to describe and analyze the software architecture based on various QAs. It was stated that software architectural analysis could help detect software defects in the early phases of software development, reducing the overall cost. The method performed the software analysis using three different dimensions, viz., the software’s functionality, the structure or design of the software, and the allocation choices made to embed various functionalities in the system. The analysis assesses the architectural assumptions made while making various design choices in developing the software and the inherent risks involved. SAAM evaluates software by analyzing it in various scenarios and determining if a particular scenario requires any architectural changes to the software. The scenario development is performed by various stakeholders, who identify the viable scenarios. The scenarios that require architectural changes are indirect, and those that do not require any changes are called direct. SAAM expresses various scenarios based on the modifications required and associates them with the cost of respective modifications. The costs serve as the basis of the overall evaluation of the candidate architectures. SAAM has been validated in various case studies, such as air traffic control, user interface development environments, Internet information systems, keyword in context (KWIC) systems, and embedded audio systems.

SAAM-founded on Complex Scenarios (SAAMCS) [Lassing et al., 1999] was an extension of SAAM, introduced to emphasize the complexity of risk assessment scenarios. SAAMCS majorly focuses on the flexibility perspective, unlike SAAM, which focused on modifiability. Like SAAM, stakeholders play an essential role in SAAMCS and act as initiators of scenarios. Software Architectural Description in SAAMCS is partitioned into microarchitecture and macroarchitecture. SAAMCS introduces and uses a measurement instrument to express the effect of scenarios, which comprises:

1. four levels of impact of the scenario,

2. the number of owners involved in the information system, and

3. four levels regarding the presence of version conflicts.

SAAMCS has been evaluated for the business information systems.

Extending-SAAM by Integration in the domain (ESAAMI) [Molter, 1999] is an improved version of SAAM, emphasizing the reuse of knowledge present in software architectures and the analysis templates. ESAAMI combines analytical and reuse concepts and integrates the SAAM in the domain-specific and reuse-based development process. ESAAMI proposes packages of analysis templates which represent the essential features of the domain. Analysis templates collect reusable products that can be deployed in the various steps of the method. These products are protoscenarios, evaluation protocols, protoevaluations, and architectural hints and weights. Protoscenarios are generic descriptions of reuse situations or interactions with the system used in the scenario development phase of subsequent architecture analyses. The method is still in the improvement process.

Software Architectural Analysis Method for Evolution and Re-usability (SAAMER) [Lung et al., 1997] is an extension of SAAM emphasizing evolution and reusability. Besides the stakeholders’ involvement, SAAMER requires additional information on changes and domain experts’ experiences. SAAMER was used to model different information types, such as stakeholder information, architecture information, quality information, and scenarios. SAAMER considers several architectural views (viz., static, map, dynamic, and resource) to analyze architectures for evolution and reusability. The analysis reveals the system’s sensitivity due to the change in or the importance of objectives and future requirements. SAAMER has been applied to the large-scale telecommunications systems for identifying the areas
for improvement and the potential for reuse.

The Architectural Trade-off Analysis Method (ATAM) [Kazman et al., 1998] provides a framework for evaluating software architecture with respect to multiple QAs, particularly, modifiability, performance, availability, and security. ATAM introduces the notion of tradeoff among multiple QAs, given a software architectural description. ATAM requires a software architectural description based on Kruten’s “4+1” views [Kruchten, 1995] and requires several views, viz., a dynamic view, a system view, and a source view. It also considers various scenarios, such as use cases, growth scenarios, and exploratory scenarios, and qualitative analysis heuristics to perform the evaluation. The evaluation process comprises various scenarios and requirements, architectural views and scenario realization, attribute model building and analysis, and tradeoffs. The attribute analysis leads to the discovery of various tradeoff points and sensitivity points. A tradeoff point is a property that affects more than one attribute and is a sensitivity point for at least one attribute. ATAM provides for an iterative improvement strategy. At the end of the evaluation process, the analysis results are compared with the system requirements. If the system-predicted behavior is reasonably close to the requirements, the designers are signaled to proceed with the detailed design phase; else, they are notified to develop an action plan for changing the architecture. ATAM has been applied to several software systems and is under research.

Scenario-Based Architectural Re-engineering (SBAR) [Bengtsson and Bosch, 1998] drives the architectural redesign focusing on reliability and the performance QAs. Similar to ATAM, SBAR also offers an iterative process for software evaluation and improvement. SBAR uses four evaluation techniques: scenarios, simulation, mathematical modeling, and experience-based reasoning. Each of these techniques is used for specific QAs. Scenarios are used for evaluating software development-based QAs, such as their maintainability and reusability. The architectural performance in various scenarios is analyzed. Simulations evaluate the software operational-based QAs, such as its performance and fault-tolerance. Mathematical models perform a static evaluation of the architectural design models and act as an alternative to simulations. Experience-based reasoning is based on subjective factors, such as intuition and experience, and uses the experts’ tacit knowledge. SBAR does not require many stakeholders’ involvement; however, the software designer acts as the evaluator. SBAR works on the detailed architectural design, and the procedure comprises:

1. defining a set of scenarios for each software quality,
2. manually executing the scenarios on the architecture, and
3. interpreting the results.

If all scenarios are executed without problems, the quality attribute of the architecture is optimal. SBAR has been validated for various software systems, such as the fire-alarm system, measurement system, and dialysis system.

Quality-Driven Architecture Derivation and Improvement (QuaDAI) [González-Huerta et al., 2013] is a metrics-based method used for derivation, evaluation, and improvement of software product architectures obtained in Software Product Line (SPL) development processes. QuaDAI performs the evaluations based on a) SPL viewpoints: functional, variability, quality, transformation, and b) a process consisting of a set of activities conducted by model transformations to allow the automatic derivation, evaluation, and improvement of a product architecture from the SPL architecture. The procedure comprises:

1. deriving the product architecture from SPL architecture to maximize the QA-requirements,
2. evaluating the product architecture by measuring the QA fulfillment as per the requirements, and
3. transforming the product architecture to meet the QA-requirements.

The authors report that QuaDAI achieves better results than ATAM when evaluated by novice software architectural evaluators, and there exists a potential to improve the method’s usability.

Limitations of the existing works:

1. Most of the existing works consider a subset of QAs for software evaluations.
2. The complexity in the existing methods limit their use. Also, there is an overhead of learning the evaluation methods’ details or training the novice software architect evaluators to work with them.
3. Almost all existing methods require inputs from stakeholders, software architects, and various experts. The dependency comes with an additional cost and limits the scope of knowledge to the involved participants’ experience.
4. To the best of our knowledge, none of the existing works recommend the necessary software tactics to achieve various QAs. Knowledge of the essential software tactics used for specific QAs is unexplored in the existing works.
To overcome the limitations listed above, we propose a fully-automated framework for software architectural evaluation that leverages the knowledge present in architectural images and the relationship between various architectural patterns and QAs to perform the evaluations. Using the relationship between various software tactics and QAs, our framework also recommends the necessary tactics to achieve specific QAs. We have not come across any work exploring such research direction for architectural evaluation to the best of our knowledge.

3 PROPOSED SYSTEM

For a given software architectural image as input, our framework detects the architectural patterns present in it, evaluates it on the basis of its QA strengths and weaknesses, and finally recommends the necessary tactics to achieve specific QAs. We have not come across any work exploring such research direction for architectural evaluation to the best of our knowledge.

3.1 Estimating the architectural patterns present in the input image

We have extracted image features, software components, and their interconnections present in the input image. Each image is matched with all the remaining images in the database for similarity. Later, these components’ labels have also been extracted, and the information is stored as a structured dataset.

Architectural Image Matching. An in-house database consisting of images of different architectural patterns and designs has been considered for this study. The images are pre-processed to remove noise and are brought to same size before extracting features. Feature extraction ensures that only the salient features from the images are considered for matching instead of comparing the whole image. We have used SIFT [Lowe, 1999] that is an interest point detector having corner properties at different scales for feature extraction. A suitable descriptor is also utilized to describe the interest points. Every image could have multiple and different numbers of interest points. Interest point descriptors across different architectural images are subjected to achieve a mutual similarity, and then the count of high matching interest points is used as a similarity measure between the two architectural images. For instance, if the images A and B having $N_a$ and $N_b$ number of SIFT interest points have $N_{ab}$ number of highly correlated descriptors in common, then the match score is computed as given in equation (1). During testing, each image in the database is matched with all other images in the database.

Input Architectural Query Image

Preprocessing

Feature Extraction

Top-k matches

Majority Voting

Class Label

Database

Preprocessing

Feature Extraction

Feature Matching (1-vs-all)

Model Parameters

TEST PHASE

TRAINING PHASE

Figure 3: Block Diagram for architecture pattern matching.
the database to obtain a dis-similarity score. The images belonging to the same architectural pattern are expected to have a low score, whereas the ones belonging to different pattern classes have a high score value. The score of matching each image with all other images is obtained and the score list is sorted in increasing order of dis-similarity. Further, for a given query architectural image, the most similar architectural image in the database is determined, and its label is given to the query image. Block diagram of the proposed architectural matching system is given in Figure 3.

\[
Score = 1 - \frac{N_{ab}}{\min(N_a, N_b)}
\]  

3.3 Mapping the architectural patterns with QAs and tactics

Table-2 lists the QAs and sub-QAs listed by ISO/ IEC 25010 standard. We treated the QAs mentioned in this table as a reference point for our study. The information about the necessary tactics required to achieve these QAs was derived by reviewing some standard software architectural reference books [Bass et al., 2003, Bachmann et al., 2003, Scott and Kazman, 2009, Bi et al., 2018, Mark, 2015, Trowbridge, 2004, Scott and Kazman, 2009, Li et al., 2020, Osses et al., 2018, Kim et al., 2009, Fielding and Taylor, 2000].

The rationale for selecting the specific QAs and tactics: We consider the complete set of QAs as defined by ISO/ IEC 25010 standard. To derive the information of tactics required to achieve the QAs, we utilized the existing literature as mentioned above.

Our framework also estimates the architectural patterns present in an input image by obtaining top-k matches from the matching score file. Instead of just referring to the Rank-1 match, top-k matches are considered, where k is a small constant manageable in further processing. The text from the top-k images having the same label as the rank-1 are extracted, and components are identified read using Optical Character Recognition (OCR). These components are then added to a Bag of Words representation that is later referred to for determining a query image’s architectural pattern. The labels of the software components present in the image are mapped to the software components present in standard architectural patterns, such as broker, pipe-and-filer, event-bus, etc. A single architecture pattern may fall into more than one class. It is then a multi-label classification problem that could be handled using bag, chain, or power-set of classifiers. Table-1 lists the patterns considered by our framework.

4 EXPERIMENTAL SETUP

4.1 Database

An in-house database consisting of different architectural patterns and designs has been collected using a web crawler. The images are collected from online available sources such as official blogs, technical write ups with ground truth. After collecting these images, we manually filtered them to remove the images not projecting the relevant design patterns or not of desired quality. Ground truth for the remaining images are manually annotated for the training. We have considered nine architectural patterns for this study that are listed in Table 1. Our architectural design image database contains 363, 51, 57, 93, 96, 100, 384, 282, 95, 75, 81, 119, 68, and 138 images respectively. In total, there are 2002 images. Images are of three channels with RGB color coding different resolutions and sizes. Some of the sample architectural images are shown in Fig. 4. Our database is shared here https://www.doi.org/10.6084/m9.figshare.14156408.

4.2 Evaluation metrics for computing similarity in images

Image matching effectiveness is measured based on the correct recognition rate (CRR), also known as rank one accuracy. It is the percentage of images, out of total images, for which the recognition is correct at Rank-1 retrieval. Suppose out of n test images, x images are found to be true
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker/Messaging Hub</td>
<td>Maintainability, Reliability, Portability, Testability, Modifiability, Integrity, Modularity</td>
<td>Increased complexity, Increased maintenance effort, Availability, Performance</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an Intermediary, Restrict Communication paths, Use Runtime registry, Abstract common services</td>
<td></td>
</tr>
<tr>
<td>Layered</td>
<td>Testability, Maintainability, Reliability, Portability, Security, Modifiability, Integrity, Modularity</td>
<td>Agility, Ease of Deployment, Scalability, Performance, Reliability, Compatibility</td>
</tr>
<tr>
<td></td>
<td>Limitation: Maintain Semantic coherence, Use Encapsulation, Use an Intermediary, Restrict Communication paths, Abstract common services</td>
<td></td>
</tr>
<tr>
<td>Event-Bus/Mediator</td>
<td>Modifiability, Usability, Performance, Scalability, Agility, Deployment</td>
<td>Compatibility, Integrity, Security, Reliability, Availability, Testability</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Anticipate expected changes, Limit possible options, Use an intermediary, Restrict Communication paths, Concurrent communication, Use start-up time binding, Parallel-Processing, Load Balancing, Failover systems, Pipelining, stream processing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Reliability, Usability</td>
<td></td>
</tr>
<tr>
<td>Pipe-and-Filter</td>
<td>Performance, Modifiability, Maintainability, Portability, Compatibility</td>
<td>Scalability, Maintainability, Modifiability</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Restrict Communication paths, Use an intermediary, Abstract common services</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Redundancy, Replication, Active-replication, passive replication, shadowing, spare, Concurrency, Sharding</td>
<td></td>
</tr>
<tr>
<td>Repository/Database</td>
<td>Reliability, Recoverability, Maturity, Availability, Fault tolerance, Portability, Usability, Compatibility, Security</td>
<td>Maintainability</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Restrict Communication paths, Use an intermediary, Abstract common services</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Redundancy, Replication, Active-replication, passive replication, shadowing, spare, Concurrency, Sharding</td>
<td></td>
</tr>
<tr>
<td>Microservices [Li et al., 2020, Osses et al., 2018]</td>
<td>Maintainability, Scalability, Agility, Testability, Deployability, Resilience, Reliability, Usability, Security, Portability, Performance, Scalability, Modifiability, Integrity, Modifiability, Performance, Monitorability</td>
<td>Use Encapsulation, Abstract common services, Restrict Communication paths, Use an intermediary, Development and Operations (DevOps), Concurrency, Profiling, Redundancy</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Use Encapsulation, Abstract common services, Restrict Communication paths, Use an intermediary, Development and Operations (DevOps), Concurrency, Profiling, Redundancy, Use an intermediary, Raise the abstraction level</td>
<td></td>
</tr>
<tr>
<td>Model-View-Controller</td>
<td>Maintainability, Performance, Modifiability, Modifiability, Usability, Resilience, Availability, Security, Reliability, Portability</td>
<td>Compatibility</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding</td>
<td></td>
</tr>
<tr>
<td>Peer-to-Peer</td>
<td>Availability, Scalability, Resilience, Reliability, Performance, Recoverability</td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Use Encapsulation, Abstract common services, Restrict Communication paths, Use an intermediary, Development and Operations (DevOps), Concurrency, Profiling, Redundancy, Use an intermediary, Raise the abstraction level</td>
<td></td>
</tr>
<tr>
<td>Presentation Abstraction Controller</td>
<td>Maintainability, Reliability, Portability, Usability, Security, Modifiability, Testability</td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Use Encapsulation, Abstract common services, Restrict Communication paths, Use an intermediary, Development and Operations (DevOps), Concurrency, Profiling, Redundancy, Use an intermediary, Raise the abstraction level</td>
<td></td>
</tr>
<tr>
<td>Client-Server</td>
<td>Modifiability, Maintainability, Security, Portability, Security</td>
<td>Reliability</td>
</tr>
<tr>
<td></td>
<td>Strength: Authentication, Authorization, Restoration, Ping/Echo, Checkpoint, Rollback, Shadow, State Re-synchronization, Voting, Synchronization, Transaction, Removal from Service, Sampling Control Frequency, Data Integrity, Usage Integrity, Maintain multiple copies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Profile, Logging, Concurrency, Dynamic Priority Scheduling, Data Replication</td>
<td></td>
</tr>
<tr>
<td>Space-Based [Kim et al., 2009]</td>
<td>Agility, Deployability, Performance, Scalability</td>
<td>Reliability, Testability, Modifiability, Maintainability</td>
</tr>
<tr>
<td></td>
<td>Limitation: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding, Statelessness, Data Replication</td>
<td></td>
</tr>
<tr>
<td>Representational State Transfer (REST) [Fielding and Taylor, 2000]</td>
<td>Availability, Modifiability, Testability, Performance, Visibility, Modifiability, Reliability, Portability, Flexibility, Scalability, Simplicity</td>
<td>Usability</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding, Statelessness, Data Replication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Use Encapsulation, Abstract Common Services, Priority-queue scheduling, Wildcard subscribers, Bi-directional communication, subscription handling</td>
<td></td>
</tr>
<tr>
<td>Publisher-Subscriber [Trowbridge, 2004]</td>
<td>Modifiability, Scalability, Reliability, Performance, Integrity, Testability, Usability</td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Strength: Maintain Semantic coherence, Use Encapsulation, Use an intermediary, Use Runtime Binding, Statelessness, Data Replication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Limitation: Use Encapsulation, Abstract Common Services, Priority-queue scheduling, Wildcard subscribers, Bi-directional communication, subscription handling</td>
<td></td>
</tr>
</tbody>
</table>
matches at Rank-1, then:

\[ CRR = \left( \frac{x}{n} \right) \times 100 \]  \hspace{1cm} (2)

\textbf{The rationale for choosing CRR:} For a given query image, CRR represents the most similar item in the database at rank-1, which corresponds to the best identification and suitable label for the query.

## 4.3 Experimental setting and results for image matching

Architectural images in our database are scaled to the size 100 × 100. We followed the one-vs-rest matching strategy where each image in the database is matched with all the remaining images to obtain the matching score. Since we have 2002 images in the database, we obtained 2036329 scores that contains 391525 genuine and 1644804 imposter matching scores. Our framework achieves a Correct Recognition Rate (CRR) of 62.65% in identifying the architectural patterns when all the classes have been considered. However, CRR keeps increasing when fewer classes are considered and reaches 98.46% when architectural images of any two classes are only compared. The change in CRR value with change in the number of classes is annotated and depicted using graph shown in Figure 5. Here, the classes represent the architectural patterns (nine) considered by us in this work.

### 5 EFFICACY EVALUATION

The proposed system provides QA analysis for a given architectural diagram of a software system. To evaluate the proposed framework’s effectiveness, we conducted experiments by sharing the generated QA analysis with experimental group participants and compared the performance with control group participants. As the part of the experi-
<table>
<thead>
<tr>
<th>QAs</th>
<th>Categories of tactics</th>
<th>Tactics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Resource Demand</td>
<td>Increase Computational Efficiency, Reduce Computational Overhead, Manage Event Rate, Control Frequency of Sampling Events, Bound execution times, Bound queue sizes, Control Demand of resources</td>
</tr>
<tr>
<td></td>
<td>Resource Management</td>
<td>Increase Physical Concurrency, Balance Resource Allocation, Maintain multiple copies of data or computation</td>
</tr>
<tr>
<td></td>
<td>Resource Arbitration</td>
<td>Increase the Logical Concurrency, Determine the appropriate Scheduling Policy, Use Synchronization protocols</td>
</tr>
<tr>
<td>Availability</td>
<td>Fault Detection</td>
<td>Ping or Echo, Heartbeat, Exceptions</td>
</tr>
<tr>
<td></td>
<td>Fault Recovery</td>
<td>Voting, Active redundancy (using synchronization), Passive redundancy (using warm restart, dual redundancy, triple redundancy, Spare, Shadowing, State Re-synchronization, Checkpoints or Rollbacks)</td>
</tr>
<tr>
<td></td>
<td>Fault Prevention</td>
<td>Removal from service, Transactions, Process Monitor</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Localize expected modifications</td>
<td>Maintain semantic coherence, Isolate the expected change, Raise the abstraction level, Limit the possible options, Abstract common services in the primary modules</td>
</tr>
<tr>
<td></td>
<td>Restricting visible responsibilities</td>
<td>Hide information, Maintain existing interface, Restrict communication, Use an intermediary</td>
</tr>
<tr>
<td></td>
<td>Defer Binding time</td>
<td>Runtime registration, Configuration files, Polymorphism, Component replacement, Adherence to the defined protocols</td>
</tr>
<tr>
<td></td>
<td>Preventing the Ripple effect</td>
<td>Break the dependency chain (using virtual machine, repository, name-server, dynamic scheduling algorithm, publisher-subscriber pattern), make the data self-identifying, limit communication paths</td>
</tr>
<tr>
<td>Security</td>
<td>Resisting attacks</td>
<td>Authenticate, Authorize, Maintain Data Confidentiality, Maintain Data Integrity, Limit Exposure, Limit Access</td>
</tr>
<tr>
<td></td>
<td>Detecting attacks</td>
<td>Intrusion detection</td>
</tr>
<tr>
<td></td>
<td>Recovering from an attack</td>
<td>Restoration - using the tactics mentioned in the Fault Recovery category Identification using Audit trail</td>
</tr>
<tr>
<td>Testability</td>
<td>Manage Input and Output</td>
<td>Record or Playback, Separate interfaces for implementation, Specialized access routines or interfaces</td>
</tr>
<tr>
<td></td>
<td>Internal Monitoring</td>
<td>Built-in Monitors</td>
</tr>
<tr>
<td>Usability</td>
<td>Separate User-Interface</td>
<td>Use Semantic Coherence, Maintain multiple views</td>
</tr>
<tr>
<td></td>
<td>Support User Initiative</td>
<td>Cancel, Undo, Aggregate</td>
</tr>
<tr>
<td></td>
<td>Support System Initiative</td>
<td>User model, System model, Task model</td>
</tr>
</tbody>
</table>
ment, we ask the following questions:

1. **Accuracy**: How accurately experimental group participants performed in terms of correctness of QA analysis as compared to the control group participants?

2. **Time Performance**: Is there any improvement in terms of time taken to analyze a given software architectural diagram by the experimental group participants as compared to control group participants?

3. **Explainability**: How much appropriate reasoning/explanation were provided for the analysis by the experimental group as compared to the control group participants?

4. **Robustness**: How many variety of architectural patterns can be processed more accurately with the help of the proposed system by the experimental group as compared to the control group participants?

5. **Procedure**

As part of our experiments, we distributed various architectural diagrams to the controlled group (20 candidates, not having access to our framework) and experimental group (20 candidates, used our framework). We had a mixture of candidates from final year computer science undergraduates (10), postgraduates (5), and the IT industry professionals (5) with experience between 0 and 6 years in both the groups. Each of the participants had prior knowledge of the ISO/IEC 25010 Software Product Quality model and was given 28 architectural diagrams, two from each of the nine categories. We asked them to identify the significant architectural components, patterns, and overall system properties (quality attributes and tactics) perceived by them after analyzing the architectural diagrams without any additional documents. The participants were free to use any other information source available to them with appropriate citations. We manually evaluated all the assignment responses and scored them based on the following parameters for each architectural diagram to measure the accuracy and explainability:

1. Number of components and their characteristics correctly identified
2. Number of connections and their characteristics correctly identified
3. Number of design patterns correctly identified with justification
4. Number of quality attributes correctly identified with justification
5. Number of architectural tactics correctly identified with justification

We also asked each participant to share the time spent on the complete assignment to measure the time performance.

### 5.2 Results and Discussion

Each of the box plot graphs shown in Fig. 6-7 represents the minimum value, 25 percentile, 75 percentile, and maximum values, respectively.

#### 5.2.1 Accuracy and Explainability

The responses of each candidate and inference outcome of the automated model were evaluated by teaching assistants manually for their accuracy. As depicted in Fig. 6, it is evident that the usage of our proposed model to evaluate the architectural designs reduces the variation in the results and, at the same time, increases the overall correctness of the responses. The explainability score indicates that the experimental group participants are performing better than the controlled group participants.

#### 5.2.2 Time Performance

The box plot in Fig. 7 clearly depicts that the experimental group performed far better than controlled group and that too with higher accuracy in their responses. It shows that using the proposed approach will help novice and experienced candidates in effectively evaluating the architectural designs with limited variation in their responses in terms of time, efforts, and accuracy.
5.2.3 Robustness

We included the robustness parameter to avoid bias due to candidates’ prior experience in the control and experimental groups. Candidates might be good at evaluating the architectural patterns that they have worked with within the past. This parameter analyzed whether a candidate could answer with higher accuracy across the nine different categories of images. To measure this parameter, we aggregated each candidate’s score for each category and calculated the percentage of categories for which they have scored more than 75 percent. As shown in Fig. 8, the experimental group shows a better hold on robustness than the controlled group.

6 THREATS TO VALIDITY

Several threats affect the validity of this work. Firstly, the scope of our work is limited only to the considered set of architectural patterns. Our framework cannot detect the patterns not considered currently and thus does not comment about their QAs or tactics. However, this can be overcome by extending our study for the left-out patterns.

Secondly, the images have been acquired from the web, which calls for many challenges, such as size or resolution variation, illumination variation, blurriness, noise, and color in the images, affecting the overall process; because feature extraction is primarily dependent on the quality of the images. Therefore, manual filtering for removing the images not projecting the considered patterns and are of low quality is performed. Further, the pre-processing of acquired images is done in order to enhance them for better feature extraction. Another threat could be the use of OCR in the mapping procedure. The images can have labels written in various font styles and sizes, or it could be the case that some images may contain handwritten labels. Many independent variables were used in these models, which could affect the accuracy of predictions. However, we used stepwise techniques to identify the optimum number of variables for the models and tested multi-col-linearity models. These risks were partially mitigated using a classic N-crossfold validation to evaluate the models and demonstrate that they were generalizable.

Lastly, the experiments conducted with the controlled group and experimental group may have some loopholes. We asked candidates to follow the guidelines as much as possible to ensure that equivalence among them in terms of resources available to them. The higher number of candidates should have been better to reduce noisy and biased responses from the results. We also asked the candidates to share the total time spent on solving the assignment but not for each category or architectural design diagram. It could have given more granular information about the difficulty level across different categories.

7 CONCLUSION AND FUTURE WORK

An automated framework to evaluate a software architecture with image processing, optical character recognition, and inference from the QA knowledge-base is quite useful. It reduces the variation among responses and increases the accuracy across the multiple categories of architectural patterns. Further, the effectiveness of architecture image matching could be improved by exploring graph-based techniques on a more curated image dataset that can connect vision components and determine combinatorial similarity. Possible improvements in OCR methods could be explored for better handling of design artifacts and handwritten annotations. These improvements will help increase the explainability of our framework and improve the overall model’s robustness. Developing the framework in the form of a full-fledged automated application for community use is a part of our future work. We are currently working on the prototype of the system and are planning to add more architectural knowledge artifacts to increase the effectiveness of this approach and provide more advanced recommendations.
REFERENCES


