A Rule-based System for Automated Generation of Serverless-Microservices Architecture

Conference Paper · October 2018

DOI: 10.1109/SysEng.2018.8544423

2 authors, including:

Indika Perera
University of Moratuwa
220 PUBLICATIONS **1,865** CITATIONS

SEE PROFILE

A Rule-based System for Automated Generation of Serverless-Microservices Architecture

K. J. P. G. Perera and I. Perera

Department of Computer Science and Engineering
University Of Moratuwa
Katubedda, Sri Lanka
Email: (pravinda.12, indika)@cse.mrt.ac.lk

***Abstract*—Software being ubiquitous in today's systems and business operations, it's highly important to structure the high-level architecture of a software application accordingly to deliver the expected customer requirements while accounting for quality measures such as scalability, high availability and high performance. We propose TheArchitect, a rule-based system for serverless-microservices based high-level architecture generation. In the process of auto generating serverless-microservices high-level architecture, TheArchitect will preserve the highlighted quality measures. It will also provide a tool based support for the high-level architecture designing process of the software architect. Any software developer will be able to use TheArchitect to generate a proper architecture minimizing the involvement of a software architect. Furthermore, the positives of microservices and serverless technologies have made a significant impact on the software engineering community in terms of shifting from the era of building large monolith applications containing overly complex designs, to microservices and serverless based technologies. Hence TheArchitect focuses on generating best fitted microservices and serverless based high-level architecture for a given application.**

***Index Terms*—Software Architecture, Microservices Architecture, Serverless Architecture, Rule-based Systems, Domain Specific Software Architecture**

I. Introduction

Software architecture is a field of art which continuously evolve with the latest findings in the world of software engineering. Modern day world of software engineering is shifting from designing and developing large monolithic systems towards serverless and microservices based technologies in building enterprise applications [1], [2], [3]. The traditional process of designing the high-level architecture diagram for a given system is not only tedious but also might be error prone when it is conducted by a software architect without proper expertise and experience. A promising alternative is to automate the generation of high-level architecture, providing a tool based support for a software architect in terms of simplifying and accelerating the process of designing a high-level architecture as well as assisting any software developer to generate a proper architecture, minimizing the involvement of a software architect.

Most of the related work with respect to high-level software architecture designs focus on evaluations [4], [5], [6], [7] and improvements [8] conducted on the manually generated architecture designs. Another area of research has been on Architecture Description Languages (ADLs). ADLs are useful in visualizing a designed high-level architecture. Research work also has been carried out in classifying and comparing different ADLs and developing guidelines in order to visually represent the high-level architecture [9]. Microservices architecture related research mostly cover the characteristics of microservices architecture, its technical aspects, its implementation patterns and its deployment techniques [10], [11], [12], [13]. Software engineering research community has focused on carrying out research on serverless architecture and its characteristics [2], [14] as well as designing, implementing, and assessing performance of serverless components [15]. To date no standout literature can be found on automating the generation of serverless-microservices based high-level architecture mostly due to the inability to fully capture the system requirements of a given system and the difficulty in identifying serverless-microservices to be inline with the provided system requirements,

TheArchitect obtains the system requirements via its input wizard. The obtained information will be processed into predefined set of models. Next, architecture generator processes those models incorporating the rules within the knowledge base to identify the necessary serverless-microservices, which will result in producing a serverless-microservices based high-level architecture design. In the presence of having an experienced software architect viewing the auto generated high-level architecture, the architect can suggest for certain modifications in which those will be reprocessed through the architecture generator and only if the modified architecture is within the required quality standards, the knowledge base will be updated accordingly allowing to improve the quality of the architecture designs generated in future.

Rest of the paper is organized as follows. Section II introduces microservices architecture, serverless architecture, rule-based systems, domain-specific software architecture, Backend For Frontend (BFF) and the problem statement. Proposed technique is presented in Section III. Implementation details are explained in Section IV. Section V and Section VI respectively contain architecture evaluation and performance evaluation. Future work and Concluding remarks are discussed in Section VII.

II. Preliminaries

We first explain the concepts of microservices architecture, serverless architecture, rule-based systems, domain-specific

software architecture and Backend For Frontend (BFF). The research problem is then formulated.

A. Microservices Architecture

Microservice is a independent and lightweight service which is model to perform a single responsibility. Microservices use a well-defined interface to communicate with other similar services and databases [10]. Designing microservices adhering to single responsibility principle facilitates to achieve high cohesion, agility, flexibility and scalability along with loose coupling [16] which increases the quality of the generated architecture as explained under the metric based evaluation [4]. Microservices maintains a separate database for each microservice. Flexibility in accommodating change in microservices compared to monolithic systems has become another reason for the world of software development to shift from monolithic to microservices architectures [1], [3], [12], [17].

B. Serverless Architecture

In terms of application development, serverless means that the developers do not have to focus on setting up and administering the servers in order to run their back-end applications [2], [14], [18]. Serverless infrastructure payments are subjected to actual server usage to its millisecond. Reduction in both operational and development costs, ability to delegate the responsibility of handling scalability and maintaining back-end infrastructure have been some of the major reasons that motivated the software engineering community to seriously consider shifting towards serverless paradigm [14], [18], [19], [20].

C. Rule-based Systems

A rule-based system is a special type of expert system, consisting of a set of rules. Rule-based systems can be built by using expert knowledge or learning from real data [21]. The ability to learn from real data allows the knowledge base of the rule-based system to continuously update [22], [21]. Most importantly, rule-based systems provide the ability to capture and refine the human expertise [22], [21]. Although many different techniques have emerged in terms of developing rule-based systems, all of them share a common set of key properties as follows.

1) *Incorporate human knowledge in conditional if-then rules*
2) *Skill level increases at a rate proportional to the enlargement of its knowledge base*
3) *Solves a wide range of possibly complex problems by selecting relevant rules and then combining the results in appropriate ways*
4) *Adaptive determination of the best sequence of rules to execute*
5) *Arrive at conclusions by retracing its actual lines of reasoning and translating the logic of each rule employed into natural language*

If-then rules, will evaluate the input it receives against the condition it has and determine the possible output. As more data comes in, these rules will continuously get updated, in which the possible output will change with time progression and result in improving the accuracy of the overall system. Furthermore, computational logic is also effectively used within rule-based systems in terms of making decisions. Deterministic logic, Fuzzy logic, Probabilistic logic and Rough logic are some of the frequent used computational logic types within rule-based systems [21].

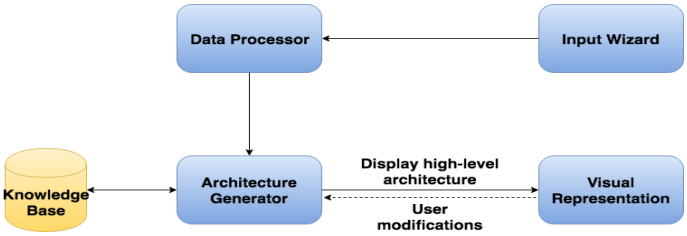

Fig. 1: High-level architecture of the TheArchitect.

D. Domain Specific Software Architecture (DSSA)

Domain-Specific Software Architecture (DSSA) is defined as an accumulation of software components, specialized for a particular type of domain, generalized for effective use across that domain, composed in a standardized structure effective for building successful applications [23].

DSSAs, is appropriate when prior experience and prior architectures are able to strongly influence on systems to be developed in future [24], [25]. The key notion is that if a considerable work has happened within a particular application domain, it is likely that a best general solution for applications within that domain will have been identified. Future applications in that domain can leverage this knowledge, with those applications having their fundamental architecture determined by the architectures of the past generations of applications [23].

E. Backend For Frontend (BFF)

BFF refers to the concept of developing dedicated server side backends for each user experience. Fundamentally the BFF is tightly coupled to a specific user experience, and will typically be maintained by the same team which develop the user interface [26].

F. Problem Statement

TheArchitect provides a rule-based system for automated generation of serverless-microservices based high-level architecture design diagrams. Hence, the problem that this research attempts to address can be formulated as follows:

How to automate, the process of generating high quality serverless-microservices based high-level architecture design diagram?

III. Proposed Technique

TheArchitect, comprises of four main components along with a knowledge base as shown in Fig. 1. The proposed technique is explained based on the high-level architecture diagram generated for Commission Calculator system as shown in Fig. 2.

A. Input Wizard

The Input wizard will request for the application domain which the high-level architecture is intended to be generated (e.g., Commission Calculator system belongs to the finance domain). Secondly the wizard will request for individual system requirements along with the belonging feature category. Finally, it will request for the following information for each listed system requirement.

API availability — Whether the focused system requirement consumes any external APIs (e.g., Desktop application of Commission Calculator system consumes finance API to fetch financial details).

Data storage necessity — Whether the focused system requirement is in the necessity for a dedicated persistent data storage.

Read / Write privileges — Information regarding the responsibility of each requirement or service with respect to read, write data to respective persistent storage.

Client application/s — The client application/s which contains the focused system requirement as a part of its functionality (e.g., services related to finance information will only be necessary for the desktop client application whereas commission grid services will be necessary for both the desktop and mobile client).

Furthermore, the user provided application domain information determines the set of rules used within the architecture generator in terms of generating serverless-microservices high-level architecture. In accordance with DSSAs [23] TheArchitect maintains a separate set of rules for each application domain. If no prior changes had been done for the existing rules set aligned with the focused application domain, TheArchitect will be consuming the initial set of rules as shown in Fig. 3.

B. Data Processor

Data Processor component maps the information obtained through the input wizard into predefined set of models. The architecture generation process within TheArchitect becomes domain-independent as well as much more optimize process due to mapping any application related requirements belonging to any domain into a predefined set of models. Below are the predefined set of models and the information mapped to those models.

Application model — Information containing the types of client applications the entire system requires (e.g., Desktop client and a mobile client for the Commission Calculator system).

Service model — Information regarding each system requirement, the feature category which the requirement belongs to and the client application/s which consumes each service (e.g., Fetching financial details and updating financial details are two system requirements for the Commission Calculator system which are categorized under finance feature category and only consumed by the desktop client).

Data store model — Information regarding the availability of APIs, persistent data storage facilities and its read/write privileges with respect to each requirement or service.

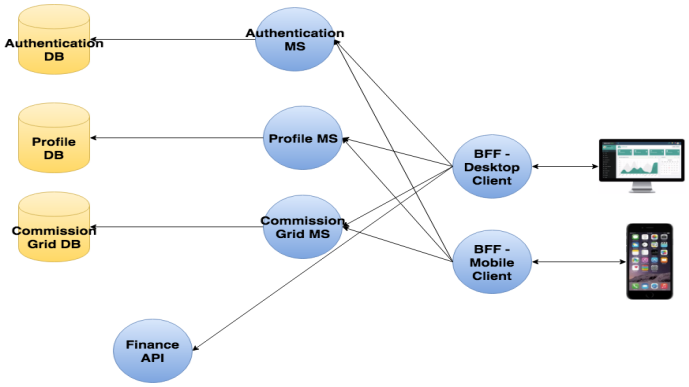

Fig. 2: High-level architecture design diagram for Commission Calculator.

```
{
  "modifiedByUser": false,
  "enableMSwithoutDB": false,
  "enableOneBFFperClientApp": true,
  "enableCombineMSs": false,
  "modifiedComponents": []
}
```

Fig. 3: Initial set of rules within the knowledge base for any application domain.

C. Architecture Generator

Architecture generator identifies the relevant data sources, serverless-microservices, BFFs and specific interactions among them for the intended application. The input provided to the architecture generator will be processed through the high-level architecture generation algorithm (Algorithm 1). Next, the output obtained through Algorithm 1 will be processed through flow shown in Fig. 6 in order to ensure that the processing within this module is carefully formulated to be inline with the defined set of rules under the relevant application domain within the knowledge base. Furthermore, in the presence of an experienced software architect requesting for modifications on the system generated architecture this module will reprocess the requested changes and if the modified architecture is within the acceptable quality measures, the current set of rules aligned with the relevant application domain within the knowledge base will be updated accordingly.

D. Visual Representation

Visual representation component will display the generated high-level architecture diagram in labeled components and interactions among them as shown for Commission Calculator system in Fig. 2. The serverless technology analysis on the identified components is shown in Fig. 4.

IV. Implementation

Implementation of the knowledge base, architecture generation algorithm and visual representation component will be

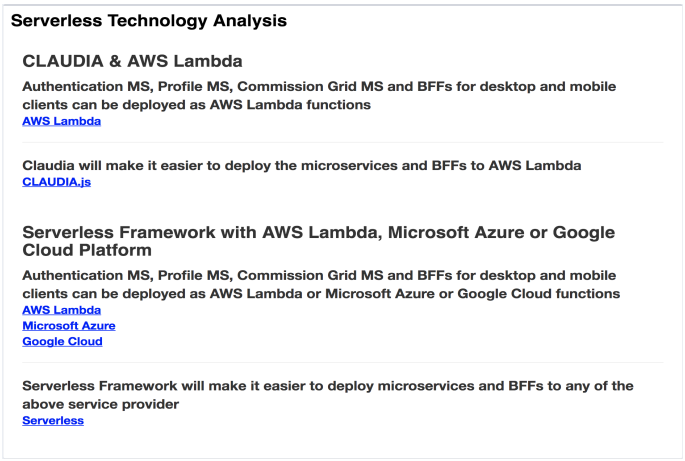

Fig. 4: Serverless technology analysis for Commission Calculator.

explained in detail with respect to automating the serverless-microservices based high-level architecture for the Commission Calculator application shown in Fig. 2 and the serverless technology analysis shown in Fig. 4.

A. Knowledge Base

The initial set of rules applied to any application domain is shown in Fig. 3. These set of initial rules will only be changed upon accepted user modification. Knowledge base contains a mapping between the application domain and a document containing the respective rules. As explained before with Commission Calculator application categorizing under the finance domain and assuming that a user modification has been accepted by TheArchitect to combine profile and authentication microservices, will result in modifying the initial set of rules under the finance domain as shown in Fig. 5.

B. Architecture Generation Algorithm

Algorithm 1 contains two parameters, namely the *services* and *dataStores* lists respectively containing service and data store models. Service and data store models contain one model object per each functional requirement within their respective lists. Algorithm 1 identifies the relevant serverless-microservices per each identified system requirement. Information related to each system requirement on the feature category it belongs to, associated client application names, APIs, database information and data read, write privileges are respectively assigned to *type*, *apps*, *apis*, *dbReadStatus* and *dbWriteStatus* from lines 10-14.

Initially, the algorithm checks the size of the *apis* list which contains the names of the associated APIs as shown in line 15, to verify whether there are any APIs associated with the focused system requirement. If there is at least on entry within the *apis* list then it will check whether there is an entry already in *apiRecords* with respect to the associated *type* for the focused system requirement as shown in line 16 if no records are existing within *apiRecords* then a new record will be added as shown in line 17 denoting that focused system requirement *type* has associated APIs. Next, in line 18 algorithm checks whether the identified *type* is already added under the *dbRecords*, meaning that there is an already added path for that specific *type* in the architecture and it needs to be cleared off and replaced, incorporating the API integration which is respectively implemented in line 19 and 20. Moreover, if there is no *dbRecords* found for the relevant *type*, algorithm will verify whether it has any records in the *recordsWithoutDbs*. If at least one record is found in *recordsWithoutDbs* the path related to that record will be cleared using Algorithm 3 and will be replaced using Algorithm 2 by adding a new path as shown in line 25.

Next, in line 29 algorithm checks whether the focused system requirement is associated with any read/write privileges with respect to any database. If the focused system requirement is associated with any read/write privileges, it will check whether previously any records has been added to then *dbRecords* with the same *type* as shown in line 30. If no related records found in *dbRecords* under the relevant *type* will add a new record to *dbRecords* and will check the whether the same *type* is recorded in *apiRecords* as shown in line 32. Here if a record is found the previously generated path will be cleared as shown in line 33. Irrespective of whether or not a *type* record found in *apiRecords* the new path will be generated and added to the *paths* list as shown in line 35.

In case if the focused system requirement not having any database read/write privileges, then line 38 will check whether a previous record is being added to the *dbRecords* under the considered requirement *type*. If a record is not found, then the relevant *type* will be added to the *recordsWithoutDbs* list as shown in line 39. Next, if a relevant entry is created within *apiRecords* then the path created with respect to the old entry will be cleared by Algorithm 3. Yet again irrespective of whether or not a *type* record found in *apiRecords*, the updated path will be generated by Algorithm 2 and will be added to the *paths* list as shown in line 43.

The above explained process, from line 7 to 46 will happen until all the functional requirements are processed. At the end of Algorithm 1 the *paths* list containing all the relevant paths of the serverless-microservices based high-level architecture will be returned. Next the returned *paths* list will be processed through the specified flow in Fig. 6. This process will retrieve the associated rules document with the current application domain and check the status of the *modifiedByUser* field. Only if a prior modification has happened it will check whether there are any components to be combined (which is denoted by *enableCombineMSs* flag status) and if it can find any matching component having less number of interactions more than which is specified under *maxInteractions* then that component will be combined with any of the previously combined components listed under that component within the rules document. Next it will check on the *enableMSwithoutDB* flag and if its value is *true* then any of the flows without a microservice will be introduced with a microservice, decoupling the API layer and BFF components. The *enableOneBFFperClientApp* flag will be only considered within the visual representation module in which if it is set to true the general treatment

```
{
   "modifiedByUser": true,
   "enableMSwithoutDB": false,
   "enableOneBFFperClientApp": true,
   "enableCombineMSs": true,
   "modifiedComponents": [
      {
         "type": "Profile MS",
         "maxInteractions": 1,
         "combinedWith": ["Authentication MS"]
      }
   ]
}
```

Fig. 5: Updated set of rules within the knowledge base for the finance domain.

of having one BFF per client application will apply or else multiple client applications will be treated by single BFF. At the end of this process the modified *paths* list will be provided to the visual representation module.

C. Visual Representation

The high-level architecture consists of named components and interactions among them as shown in Fig. 2. Visual representation module will iterate through each path model within the *paths* list. The *paths* list will contain one path model per each feature category, resulting in drawing a complete path for each feature category. Initially, visual representation technique will identify the number of application clients the focused feature category is associated with and based on that BFF component/s will be designed and linked to the path. Then, the microservices status and database status associated with the focused system requirement will be evaluated and relevant components will be drawn. If any existing APIs are associated with the relevant feature category, they would be integrated to the design accordingly. Furthermore, the serverless technology analysis will be generated by associating the identified microservices and BFFs with current facilitation for serverless implementations as shown in Fig. 4. This module also provides the ability to an experienced software architect to request any changes to the auto generated high-level architecture. Architect is free to state the component that he needs to alter and then to state the two components that becomes the predecessor and successor of it with the requested change.

V. Architecture Evaluation

Here we provide a metrics based evaluation [4] to qualitatively evaluate the auto generated serverless-microservices based high-level architecture diagrams for two real world business applications. The auto generated serverless-microservices based high-level architecture will be evaluated based on following metrics [4].

Coupling — This metric looks on the degree of dependency of one component to another.

$$Coupling = \frac{\Sigma\ Number\ of\ components\ called\ for\ each\ service}{Number\ of\ services} \quad (1)$$

Cohesion — This metric identifies the relatedness of the facilitated use cases by each component.

Number of services provided by a component — This allows to evaluate distribution of functionality over the entire design.

Algorithm 1 High-level Architecture Generation Algorithm

```
1: procedure ArchitectureGenerator(services, dataStores) ▷ services a list containing service model objects and dataStores a list containing data store model objects
2:     dbRecords ← {}
3:     apiRecords ← {}
4:     recordsWithoutDbs ← {}
5:     paths ← {}
6:     path ← {}
7:     for i ← 0 to services.size() do
8:         service ← services.get(i)
9:         dataStore ← dataStores.get(i)
10:        type ← service.getFeatureCategory()
11:        apps ← service.getAppNames()
12:        apis ← dataStore.getAPINames()
13:        dbReadStatus ← dataStore.getReadStatus()
14:        dbWriteStatus ← dataStore.getWriteStatus()
15:        if apis.size()>0 then
16:            if !apiRecords.contains(type) then
17:                apiRecords.add(type)
18:                if dbRecords.contains(type) then
19:                    paths = ClearPath(type, paths)
20:                    paths.add(PathBuilder(type, apps, apis, true, true))
21:                else
22:                    if recordsWithoutDbs.contains(type) then
23:                        paths = ClearPath(type, paths)
24:                    end if
25:                    paths.add(PathBuilder(type, apps, apis, false, false))
26:                end if
27:            end if
28:        end if
29:        if dbReadStatus∨dbWriteStatus then
30:            if !dbRecords.contains(type) then
31:                dbRecords.add(type)
32:                if apiRecords.contains(type) then
33:                    paths = ClearPath(type, paths)
34:                end if
35:                paths.add(PathBuilder(type, apps, apis, true, true))
36:            end if
37:        else
38:            if !dbRecords.contains(type) then
39:                recordsWithoutDbs.add(type)
40:                if apiRecords.contains(type) then
41:                    paths = ClearPath(type, paths)
42:                end if
43:                paths.add(PathBuilder(type, apps, apis, false, false))
44:            end if
45:        end if
46:    end for
47:    return paths ▷ generated set of paths will be returned
48: end procedure
```

Algorithm 2 Path Builder Algorithm

```
1: procedure PathBuilder(type, apps, apis, msStatus, dbStatus) ▷ type contains the feature category of the service, apps a list containing application names, apis a list containing API names, msStatus a boolean value containing microservice necessity and dbStatus a boolean value containing database necessity
2:     path ← {}
3:     path.setFeatureCategory(type)
4:     for j ← 0 to apps.size() do
5:         path.getAppNames().add(apps.get(j))
6:     end for
7:     for k ← 0 to apis.size() do
8:         path.getAPINames().add(apis.get(k))
9:     end for
10:    path.setMicroserviceStatus(msStatus)
11:    path.setDatabaseStatus(dbStatus)
12:    return path ▷ generated path will be returned
13: end procedure
```

Algorithm 3 Clear Path Algorithm

```
procedure ClearPath(type, paths)  ▷ type contains the feature category of the service and paths contains the paths list
    path ← {}
    for i ← 0 to paths.size() do
        if paths.get(i).getFeatureCategory()=type then
            paths.remove(i)
        end if
    end for
    return paths                  ▷ modified set of paths will be returned
end procedure
```

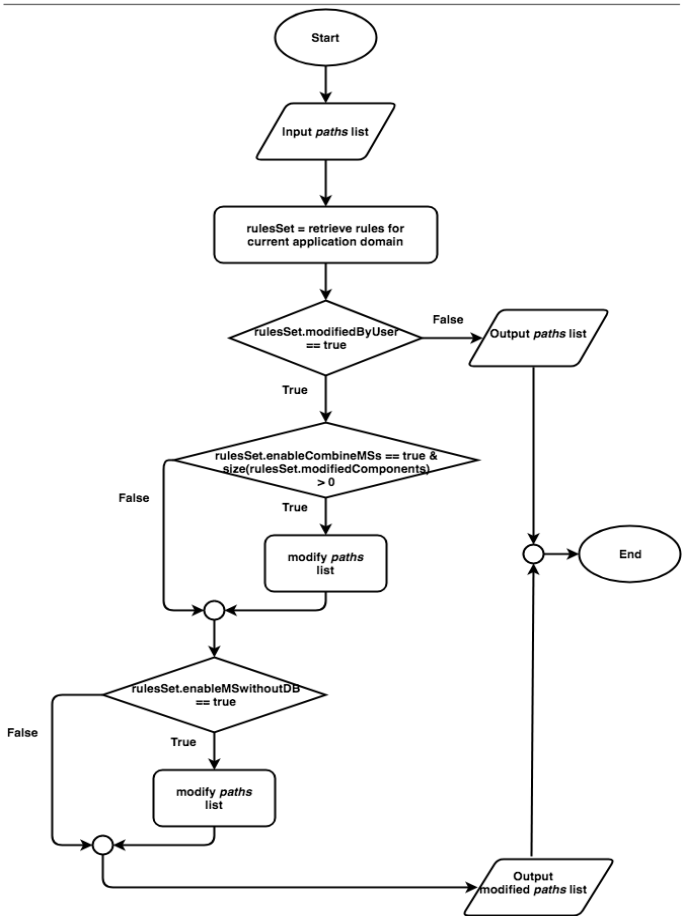

Fig. 6: Flow chart on incorporating knowledge base rules set in generating high-level architecture.

Fan in — A component with a high fan in value might possibly be a bottleneck in terms of scalability.

$$\textit{Fan in} = \textit{Number of called services of a component} \quad (2)$$

Fan out — Lower fan out value depicts lower dependency on other components.

$$\textit{Fan out} = \textit{Number of service calls of a component} \quad (3)$$

Depth of scenario — This metric provides an understanding on the complexity of a given scenario with respect to the designed architecture. Average depth of scenario is calculated as:

$$\frac{\Sigma\ \textit{Number of Component interactions for each service}}{\textit{Number of services}} \quad (4)$$

If any outlying metric value is found for a specific component, it will be identified as a problem element. We identify outlying values as the values that differ more than two times the standard deviation from the mean value [4]. Furthermore, since technically BFF is tightly coupled to a specific client application and serverless technology analysis being a mapping between the identified components against the current facilitation for serverless implementations, both of these will not be considered within the evaluations.

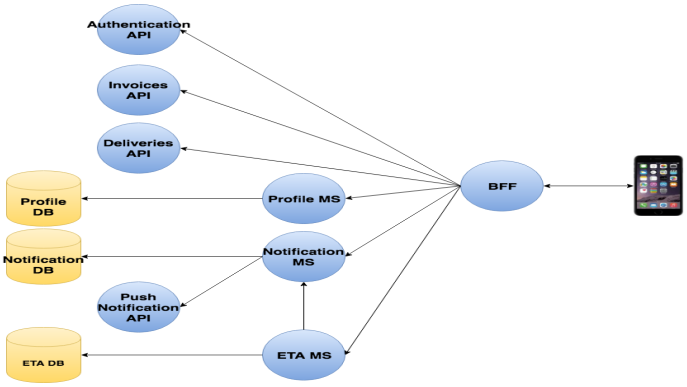

Fig. 7: High-level architecture design diagram for Order Receive application.

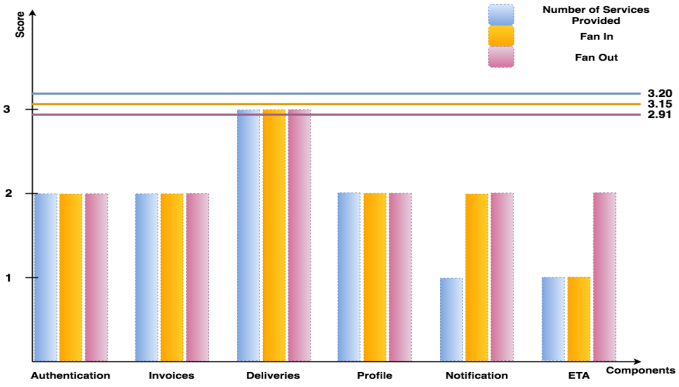

Fig. 8: Component based metrics evaluation for Order Receive application.

A. Order Receive Application

Order Receive is a mobile application developed for restaurant owners to view information on invoices and associated deliveries, along with maintaining profile based preferences as well as to receive Estimated Time of Arrival (ETA) push notifications on respective deliveries. The auto generated serverless-microservices based high-level architecture for the Order Receive mobile application is shown in Fig. 7. Metrics based evaluation for the Order Receive application is conducted among 6 identified components serving 11 high-level system requirements. The services based metrics evaluation is shown in Table I along with the component based evaluation statistics for the Order Receive application is shown in Fig. 8.

B. Inventory Management Application

Inventory Management application consists of a mobile client and a desktop client. The desktop client is for a manufacturing company which provides the ability to record sales and

TABLE I: Services based metrics evaluation for Order Receive application.

Metric	Value
Coupling	1.27 components called per service
Depth of scenario	1.64 component interactions per service

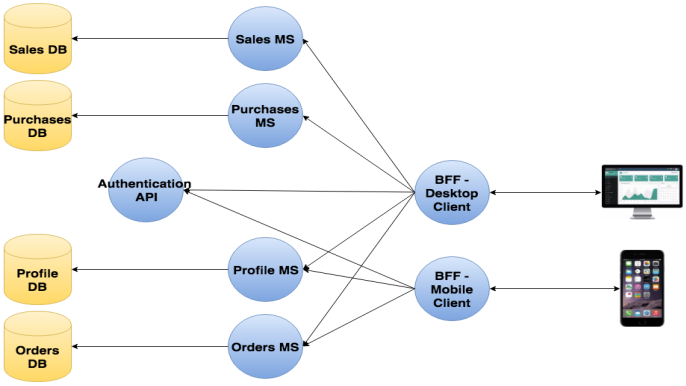

Fig. 9: High-level architecture design diagram for Inventory Management application.

purchases, as well as place orders. Mobile client is developed for vendors who provides materials for the manufacturing company. Mobile client provides the ability to check on orders, accept or reject order requests. Both clients have the ability to configure profile based preferences. The auto generated serverless-microservices based high-level architecture for the Inventory Management application is shown in Fig. 9. Metrics based evaluation for the Inventory Management application is conducted among 5 identified components serving 14 high-level functional requirements. The services based metrics evaluation is shown in Table II along with the component based evaluation statistics for the Inventory Management application is shown in Fig. 10.

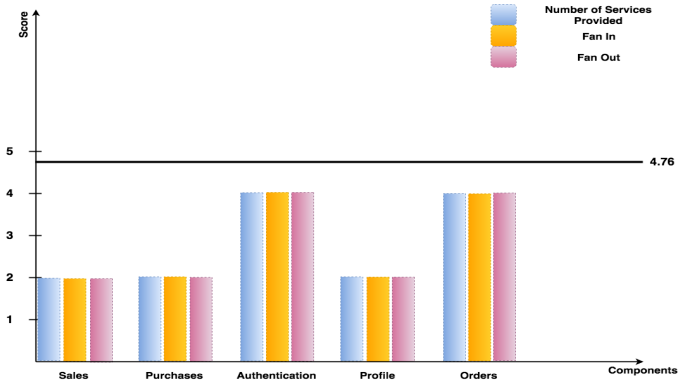

Fig. 10: Component based metrics evaluation for Inventory Management application.

TABLE II: Services based metrics evaluation for Inventory Management application.

Metric	Value
Coupling	1 components called per service
Depth of scenario	1.71 component interactions per service

TABLE III: Number of system requirements vs processing time of TheArchitect.

Number of system requirements	Processing time (seconds)
14 (Inventory Management Application)	1
33	2.3
57	4.2
93	6.8
113	8.2

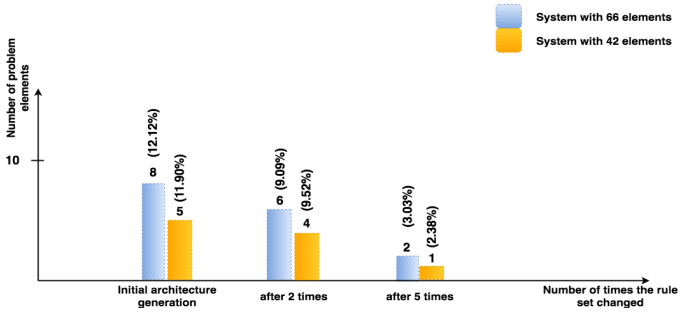

Fig. 11: Number of problem elements against the number of modifications for the rule set

C. Metric Based Evaluation Analysis

The coloured vertical bars in Fig. 8 and Fig. 10 depicts three labeled metric values per each identified component. The horizontal bar/s shown in Fig. 8 and Fig. 10 denotes the addition of the mean value and two times the standard deviation value, for each metric.

In terms of considering the component based metrics evaluation on both the applications, except the deliveries component with respect to fan out metric in the order receive application, every other component is considerably below than the outlying value with respect to of all three metrics. Next, in terms of the services based metrics evaluation on both applications, except in sending ETA push notifications, for all other services there is exactly only one interaction between the BFF and the relevant components. This results in achieving very low coupling values between individual components. This also proves the fact that the generated architecture comprises with highly cohesive components. The simplicity, maintainability and re-usability of the components is proved with recording a values less than 2 for the depth of scenario metric under both of the applications.

In summary the auto generated serverless-microservices based high-level architecture designs comprise with non-problematic, highly cohesive, loosely coupled, maintainable and reusable components.

VI. Performance Evaluation

The performance of the serverless-microservices based high-level architecture generation process is evaluated using few real world business applications which are similar to the business applications presented in Section V.

The recorded processing times in Table III indicate the complete time span taken from the inception of processing the provided system requirements up to displaying the auto generated serverless-microservices based high-level architecture. Statistics presented in Table III proves the fact that

TheArchitect has the ability to accelerate the current manual process of designing a high-level architecture. Furthermore, with the ability to process 100 system requirements, less than 10 seconds TheArchitect has proven the fact that it is capable of effectively generating high-level architecture designs for much larger, complex systems too.

In the process of evaluating the accuracy of the rule-based processing technique, Fig. 11 presents an analysis between the number of problem elements were recorded against the number of modifications conducted on the respective rule sets. Evaluation statistics on two systems respectively containing a total of 66 and 42 elements (components) are shown in Fig. 11. The statistics shows upon 5 modifications conducted on the respective rule sets, results in decreasing the number of problem elements (components) to less than 4% of the total number of elements (components).

VII. Summary and Future Work

We propose TheArchitect, to automate the serverless-microservices based high-level architecture generation process. TheArchitect also provides the ability to modify the system generated architecture. In terms of research statistics, the quality of the auto generated high-level architecture diagrams is measured against renowned quality measures. We also demonstrate a performance evaluation of TheArchitect in terms of processing time and accuracy of the rule-based processing technique.

As next steps we plan to, further enhance the level of intelligence of TheArchitect by improving the rule based processing mechanism incorporating the users decision to change the system generated architecture.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper.

References

[1] J. Lewis and M. Fowler, *Microservices*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] M. Roberts, "Serverless architectures," 2016. [Online]. Available: https://martinfowler.com/articles/serverless.html

[3] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *Proc. 11th Intl. Conf. on Internet Technology and Secured Transactions (ICITST)*, December 2016.

[4] J. Muskens, M. R. V. Chaudron, and R. Westgeest, "Software architecture analysis tool : software architecture metrics collection," in *Proc. 3rd PROGRESS Workshop on Embedded Systems*, October 2002, pp. 128–139.

[5] M. T. Ionita, D. K. Hammer, and H. Obbink, "Scenario-based software architecture evaluation methods: An overview," 2002.

[6] P. Clements, R. Kazman, and M. Klein, "Evaluating software architectures: Methods and case studies, addison wesley," 2002.

[7] R. Kazman, L. Bass, G. Abowd, and M. Webb, "Saam: A method for analyzing the properties of software architectures," in *Proc. 16th IEEE Intl. Conf. on Software Engineering*, May 1994.

[8] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef, "A two-phase process for software architecture improvement," in *Proc. IEEE Intl. Conf. on Software Maintenance*, August 1999.

[9] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," in *Proc. IEEE Transactions on Software Engineering*, January 2000.

[10] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *Intl. Journal of Open Information Technologies*, vol. 2, pp. 24–27, 2014.

[11] S. Newman, "Building microservices: Designing fine-grained systems," 2015.

[12] J. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," in *Proc. IEEE Intl. Conf. on Software Architecture Workshops (ICSAW)*, April 2017.

[13] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Proc. 9th IEEE Intl. Conf. on Service-Oriented Computing and Applications (SOCA)*, November 2016.

[14] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *Proc. IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom)*, December 2017.

[15] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. 37th IEEE Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, June 2017.

[16] C. M. Aderaldo, N. C. Mendona, and C. Pahl, "Benchmark requirements for microservices architecture research," in *Proc. IEEE/ACM 1st Intl. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, May 2017.

[17] D. Guo, W. Wang, G. Zeng, and Z. Wei, "Microservices architecture based cloudware deployment platform for service computing," in *Proc. IEEE Symposium on Service-Oriented System Engineering (SOSE)*, April 2016.

[18] A. Eivy, "Be wary of the economics of serverless cloud computing," in *Proc. IEEE Cloud Computing*, April 2017, pp. 6–12.

[19] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *Proc. 37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, June 2017.

[20] A. Avram, "Faas, paas, and the benefits of the serverless architecture," 2016. [Online]. Available: https://www.infoq.com/news/2016/06/faas-serverless-architecture

[21] H. Liu and A. Gegov, "Rule based systems and networks: Deterministic and fuzzy approaches," in *Proc. 8th IEEE Intl. Conf. on Intelligent Systems (IS)*, September 2016.

[22] F. Hayes-Roth, "Rule-based systems," in *Proc. Communications of the ACM*, September 1985, pp. 921–932.

[23] E. S. D. Almeida, A. Alvaro, V. C. Garcia, L. Nascimento, S. L. Meira, and D. Lucredio, "Designing domain-specific software architecture (DSSA): Towards a new approach," in *Proc. IEEE/IFIP Intl. Conf. on Software Architecture (WICSA)*, January 2007.

[24] J. S. Fant, "Building domain specific software architectures from software architectural design patterns," in *Proc. 33rd IEEE Intl. Conf. on Software Engineering (ICSE)*, May 2007.

[25] G. Gangyong, Z. Cuihao, and C. Wei, "A domain-specific software architecture," in *Proc. IEEE Intl. Conf. on Intelligent Processing Systems*, October 1997.

[26] S. Newman, "Backends for frontends," November 2015. [Online]. Available: http://samnewman.io/patterns/architectural/bff/