A Rule-based Toolkit for Automated Generation of Microservices Architecture

K. J. P. G. Perera
178045U

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science by Research

Department of Computer Science & Engineering
University of Moratuwa
Sri Lanka

July 2019

Declaration

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

Name: K. J. P. G. Perera

The above candidate has carried out research for the MSc Dissertation under my supervision.

Signature of the supervisor:…….. Date:
Name of the supervisor: Dr. G. I. U. S. Perera

Abstract

Software applications play a critical role in current business world; hence it is necessary to design a quality and a sound architecture which facilitates it to become a scalable, extensible and highly available solution. In terms of designing and developing software applications, software engineering community has started shifting towards serverless-microservices instead of building large monolith applications.

It requires high experience and expertise to understand each business scenario along with considering non-functional requirements too to design a high-level software architecture which would be the ground point for a software application. The traditional manual process of doing the above is tedious as well as can be error prone when architecture designing is done without proper experience and expertise, which could eventually degrade the quality of the software application.

We introduce TheArchitect, a rule-based system providing a tool-based support in order to design the best fitted high-level architecture containing serverless microservices, preserving the identified non-functional requirements too, for any given application. Furthermore, TheArchitect provides the ability to a software engineer also to generate a high-quality high-level architecture even without an experienced software architect. Considering the increasing tendency within the software engineering community to move away from monolith application development towards microservices-serverless based application development, TheArchitect has also been developed focusing on generating high-level application architecture designs based on serverless-microservices.

TheArchitect was used to generate architecture designs for restaurant management domain. System generated architecture designs for two real world applications and how experienced architects' modifications are incorporated as modified rules for future designs have been discussed. Further a performance evaluation is conducted on TheArchitect to provide an analysis on the time it takes to process the requirements and design the architecture for various real-world systems along with an industry user study is presented evaluating the usability of TheArchitect.

Keywords: Software Architecture, Microservices Architecture, Serverless Architecture, Domain Driven Design, Architecture Evaluation

Acknowledgments

I would like to express my sincere gratitude to my mentor Dr. Indika Perera and my supervisors for the guidance, support provided throughout my research. I would not be able to achieve all what I have achieved without your incredible mentorship and advices from the beginning.

I would like to thank all the staff members of the Department of Computer Science and Engineering for their continuous and generous support provided for me.

I am also grateful for all the software architects and technical leads within the industry who helped in evaluating the developed tool (TheArchitect), as well as spent their valuable time in helping out in terms of fine tuning TheArchitect.

Special thanks to my family and my loved ones for your encouragement and understanding throughout the past two years.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

HTTP	Hyper Text Transfer Protocol
API	Application Programming Interface
AWS	Amazon Web Services
ADL	Architecture Description Language
DDD	Domain Driven Design
DSSA	Domain Specific Software Architecture
BAAS	Backend-as-a-Service
MBAAS	Mobile Backend-as-a-Service
FAAS	Function-as-a-Service
BFF	Backend-for-Frontend
SOA	Service Oriented Architecture

1. Introduction

1.1 Background

In the recent past in software engineering industry there had been a huge buzz going around microservices and serverless architecture-based application development.

Microservices architecture encourages bringing up a one application consisting of tiny services, which runs its own process independently and continue to communicate among each other using lightweight communication mechanisms. Each service by nature should be independently manageable, maintainable and deployable [1]. Further, this provides the ability to use different technologies and different teams to develop and maintain different services.

Serverless architecture primarily focuses on developing applications which does not worry on managing server-side infrastructure rather it focuses on core business logic development and relying on separate service providers to maintain backend infrastructure [2], [3], [4]. Generally, these are rich client applications (web/mobile) that use the vast ecosystem of cloud access.

Traditional process of architecture generation requires the software architects to be updated on the microservices and serverless technologies and its fundamentals and then design system architecture based on those. In the world of software engineering, system requirements specification provides the business requirements of the intended system to the software architect and based on the obtained knowledge software architect will design a high-level architecture diagram. This process requires high expertise in order to understand each business scenario and design a high-level architecture diagram which would lead to develop a software application, serving identified end user requirements, preserving all noted performance measures. Most importantly this manual process is error prone as well as tedious.

1.2 Motivation for the Research

Software architecture is one prominent field which keeps on changing with latest technologies and frameworks. The current trend is to shift from architecting large monolith applications towards architecting serverless microservices [2], [5], [6]. Amazon, The Guardian, Netflix and SoundCloud are some of the popular companies which have taken up necessary directions to shift their applications towards serverless microservices [7], [8]. Further, Microsoft, Amazon Web Services (AWS), Google have heavily invested on being service providers to facilitate serverless paradigm introducing Microsoft Azure functions, google cloud functions and lambda functions.

Software architecture denotes the high-level skeleton of a software application. High-level architecture contains components, their properties and interactions among each of them. As explained, the conventional process of designing an architecture for a software application is tedious and could be error prone. In addressing the issues highlighted with the conventional process, a promising solution would be to assist software architects with tool-based support to generate high-level architecture, simplifying and accelerating the conventional process.

1.3 Research Statement

Most of the literature found on high-level software architecture designs focus on architecture improvements [7] and architecture evaluations [8], [9], [10], [11] upon the traditional process. Architecture Description Languages (ADLs) have been another high focus area under the above discipline providing literature on ways and means of visualizing high-level architecture designs. Further, some researches on ADLs have focused on comparing and classifying different ADLs and developing guidelines on visualizing designed architecture [12], [14].
Microservices related research mostly cover up the technical fundamentals, characteristics, patterns etc [1], [13], [15]. Serverless architecture related literature covers both its characteristics [2], [16] as well as its implementation aspect and performance aspect [3].
To date there is no significant literature published on development of a tool-based system to automate generation of high-level architecture designs using serverless microservices due to the difficulty of capturing the business/technical requirements and identifying the serverless microservices which suites those.

At the inception of a project producing an architecture design for a software application would require expert skills. We address the problem of needing expertise and experience in the field of software architecture to design a high-level architecture using serverless microservices, through automation. Even though field of software architecture, microservices and serverless technologies are areas in which extensive research work has been carried out, there is no major research contribution in terms of addressing the problem of eliminating the necessity of expertise and experience in the field of software architecture, to design architecture diagrams with serverless-microservices. Difficulty in both, capturing the business and technical requirements as well as identifying serverless-microservices from the provided system requirements are two major issues which happens to be the reason behind lacking any standout research contribution under this area of interest.

We propose a technique which can be used to generate a high-level architecture design for any software application using serverless-microservices.

The main research problem is as follows:

How to automate the process of producing a microservices based high-level architecture design for any given application?

1.4 Objectives of the Research

The goal of the research is to provide a tool for the architect to improve the efficiency of generating high-level architecture designs using microservices. The objectives of the research are as follows:

1. Identify the fundamentals of microservices and serverless designs
2. Identify architecture evaluation metrics to compare system generated architecture vs architects' modifications
3. Identify, design and develop the mechanism to accept system business requirements
4. Design and develop rule-based processing engine to generate the architectural components and interactions among them
5. Design and develop architecture visualization component to demonstrate the system generated architecture

1.5 Research Methodology

The research methodology comprised of research and development effort in creating a tool to auto generate high-level architecture designs using microservices.

- Initially the research starts with identifying fundamentals of both microservices and serverless based architectural styles.
- Extensive research will be carried out in various Architecture Description Languages (ADLs), architecture visualizations as well as architecture evaluation techniques.
- Research and develop a wizard to obtain system business requirements in order to understand the system context and initiate Domain Driven Design (DDD).
- Develop the rule-based processor to identify architectural components and the interactions among them
- Design and develop the visualization component, to display the system generated architecture

1.6 Contributions

TheArchitect, a rule-based toolkit was designed and developed to automate the process of generating microservices based high-level architecture diagrams as an outcome of this research. TheArchitect, supports an architect to accelerate and simplify the conventional process of designing architecture diagrams for custom applications.

Further, the following articles have been published from the research conducted so far:

- **"TheArchitect: A Serverless-Microservices Based High-level Architecture Generation Tool",** K. J. P. G. Perera, I. Perera. Published in 2018, in 17th IEEE/ACIS International Conference on Computer and Information Science (ICIS).
- **"A Rule-based System for Automated Generation of Serverless-Microservices Architecture",** K. J. P. G. Perera, I. Perera. Published in 2018, in 4th IEEE International Symposium on Systems Engineering (ISSE).

1.7 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 explains related work in the areas of microservices architecture and its principles, serverless technology and its fundamentals, rule-based systems and processing, domain specific software architecture (DSSA), Backend for Frontend (BFF) and Service Oriented Architecture (SOA) concepts, and different architecture evaluation processes.

Chapter 3 explains the research methodology breaking down to component level of TheArchitect. In depth explanations on the input wizard, data processor, architecture generator and visual representation component are captured under this chapter.

Chapter 4 explains the implementation details of the knowledge base which contains the rule set for each domain, architecture generation algorithms and flows as well as implementation information on architecture visualization.

Chapter 5 discusses the evaluation scheme for generated architecture designs as well as the experiments conducted to evaluate TheArchitect.

Chapter 6 analyzes the results and data gathered from these experiments as well as presents the major observations and findings of the research.

Accomplishments obtained out of the conducted research are listed under chapter 7. Chapter 8 contains conclusion notes and future directives on the focused research area.

2. Literature Review

2.1 Microservices Architecture

Microservices architecture counts on architecting a single application which contains collection of small services, developed independently, talking to each other using lightweight communication mechanisms, running its own process [1]. Furthermore, it allows to write different services in different appropriate languages and also managed by different teams [1], [17].

As shown in Figure 1 microservices uses a separate service which is independently developed in order to get each element of functionality whereas, monolith systems will combine all its functionalities into one single process which becomes harder to manage and maintain among different teams [17]. Furthermore, compared to monolith systems the flexibility for continuous change embracement in microservices architecture has influenced world of commercial software engineering to shift from monolith architectures to microservices based architectures. Embracing change in monolith systems is way harder compared to microservices based systems because even for a small change it needs the complete monolith to be deployed and tested. Further, due to the complexity it incurs overtime, it is harder to maintain a proper modular structure within the system [1], [5], [6].

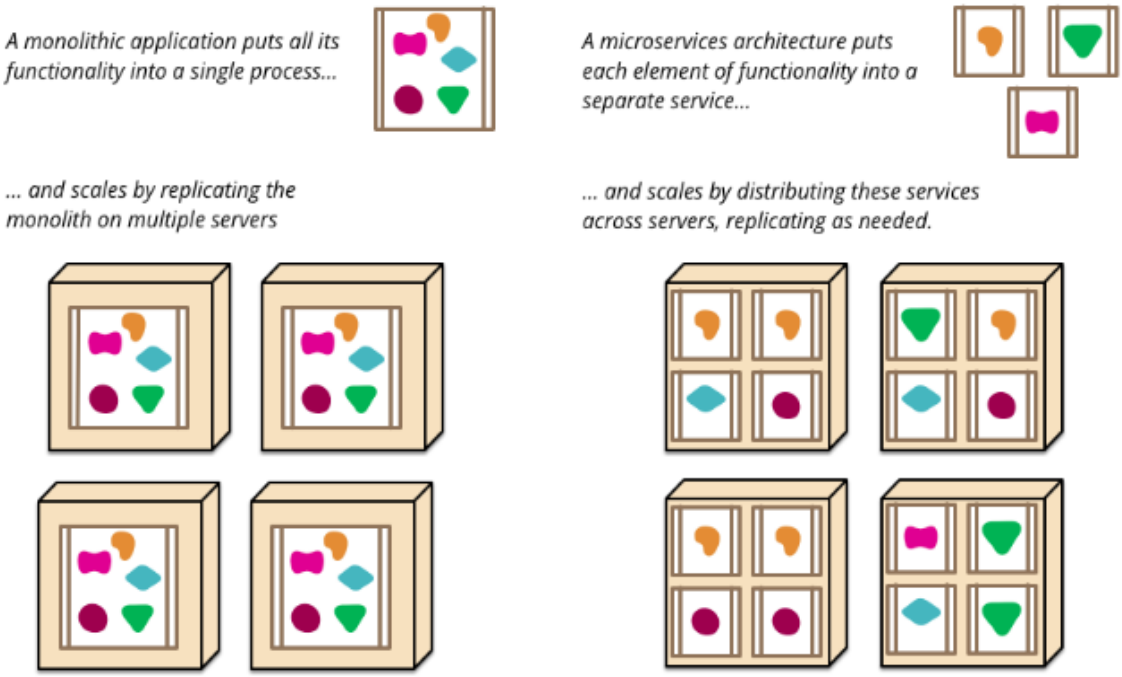

Figure 1: Monolith System vs Microservice architecture-based systems [5]

Microservices architecture defines itself based on the following characteristics [1], [7], [8], [17].

- Componentization Services as Components

  Microservices architecture is totally built on the concept of "component" which is a unit of service that is independently developed and maintained. Microservices componentization happens by breaking down into services. In today's software engineering terms external libraries tend to be considered as components but the issue is if few libraries are included in one single process, once a change done to any component requires redeployment of the entire process, whereas with services, each service is independently deployable. In this manner microservices architecture itself facilitates great support for change management process in commercialized software development.

- Work organization and segregation

Figure 2 denotes the work organization difference among the monolith architecture systems vs microservices based systems. The traditional understanding to split a large application into parts, is to different units or teams to focus on different layers of the application, leading to database teams, server-side teams and user interface design teams. The negative side of this is that even completion of a simple change might require cross team involvement.

The microservice approach being different to above is to, split up the application into services organized around business capability. Where one business capability covers user interface, server side and database implementation. Always the considered service has a business value addition.

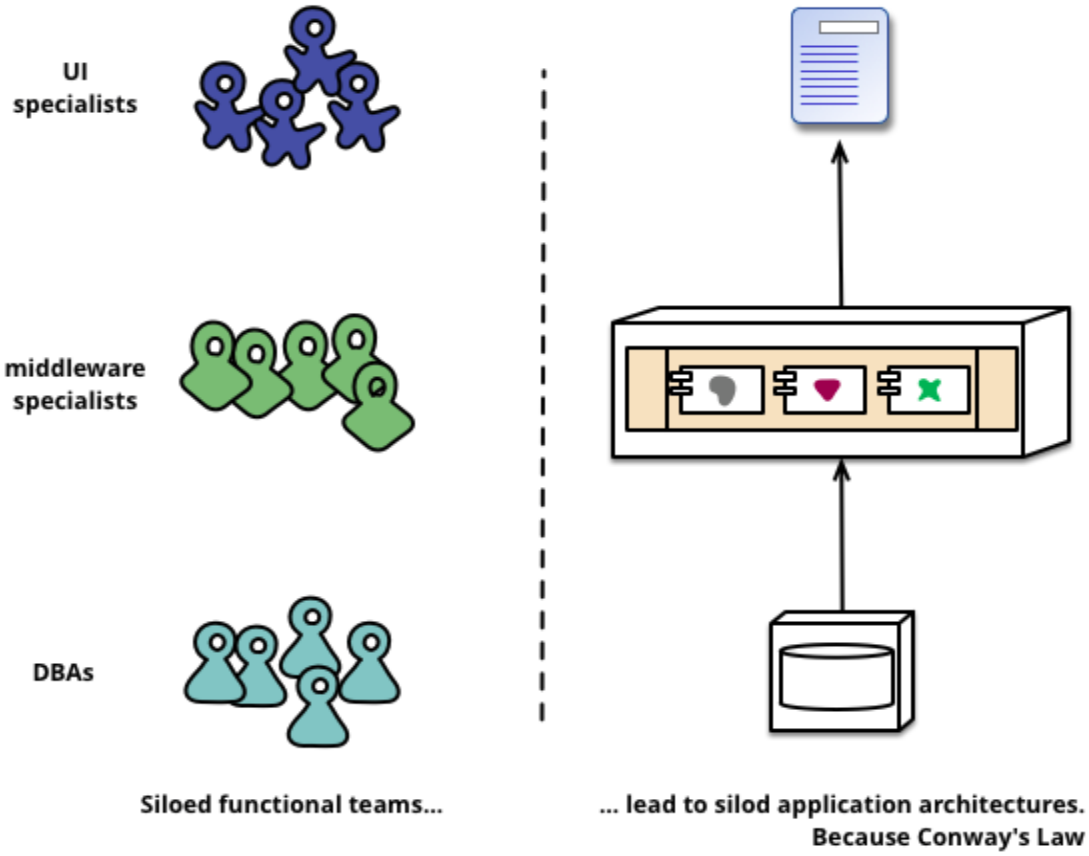

Figure 2: Work Organization for Monolith vs Microservices architecture systems [5]

- Single Responsibility

  Fundamentally, correctly identified microservices should be modeled based on the single responsibility principle. In a nutshell single responsibility principle means separate out thing that does not change for the same cause and combine any which changes for the same reason. Adhering to this principle when modeling microservices provides to have proper cohesion but at the same time having related code grouped together.

- Products not Projects

  Microservices has directly impacted commercialized software development. It tends to favor products over projects, which reflects that it does not to simply target delivering some piece of software, but it would earn ownership of an entire product life cycle covering up to the extent it would be providing support for some extent too. This has created a separate paradigm where now software engineering community thinks through the extent that software can assist to enhance the business capabilities.

- Smart endpoints and dumb pipes

  Traditional monolith systems focus on smarts communication mechanism. As an example, traditional systems encourage using ESBs, containing sophisticated messaging whereas microservices based systems does not put focus on smart communication.

  Unlike above, microservices relies on smart endpoints and dumb pipes. Its focus is on simple lightweight communication protocols. Tow most commonly used protocols are HTTP request/response facilitated with resource APIs and lightweight messaging. In terms of messaging mechanism simple implementations such as ZeroMQ or RabbitMQ is advised with microservices based applications.

- Decentralized Governance

  Microservices supports the advantages obtained by having multiple tech platforms (development approach, technologies, standards etc.) instead of having one platform for the whole system as a monolith.

  Overheads are less valued in microservices approach. In order to facilitate service contracts evolving independently service design patterns are often applied. Incorporating consumer driven patterns increase the confidence of the developed services which ensures its proper functionality.

- Decentralized Data Management

  This is another refactoring provided to the traditional monolith architecture by microservices, meaning maintaining a separate database for each microservice. This would allow to vary the chosen databases for a given application as the application is divided into different microservices.

- Infrastructure Automation – Continuous integration

  Evolution of the cloud servicing platforms such as AWS, Google Cloud, Microsoft Azure has reduced the complexity as well as time to market for applications built and deployed as microservices.

  Most of the microservices based applications are being built by agile teams, those highly respects continuous integration and delivery (CI/CD). As shown in Figure 3 high focus lies on automated test execution and automated deployment. These features have added much value in developing commercial applications in today's context which happens to be so much volatile.

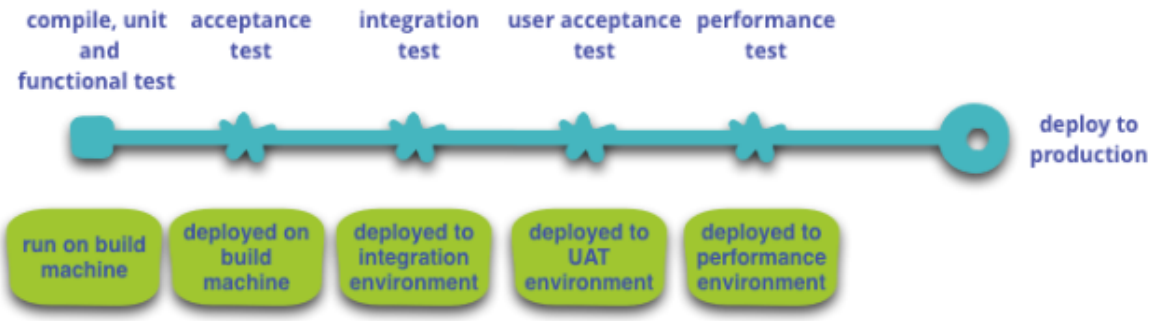

Figure 3: Basic Build Pipeline [5]

- Design for failure

  Another fundamental which should be adhered in designing true microservices are fault tolerance of its designed services. Initially, this might include additional complexity to be handled compared to monolith systems but eventually as the expansion of the system and growth happens this will increase the quality and maintainability.

- Evolutionary Design

  Microservices design provides the developers to control change without slowing down the change. In simple terms microservices embrace change without any hesitance. Furthermore, microservices design allows any particular service within the current system to be consumed by any other third party at any time and complete its need which would be difficult to accompany with a monolith system.

Apart from many advantages noted above one of the major concerns with microservices happens to be the intercommunication via Remote Procedure Calls (RPCs) which is relatively costlier compared to process calls within a monolith system. Furthermore, in terms of commercial development using microservices, increases the resource usage with the need to maintain its own container with required memory and Central Processing Unit (CPU) for each separate functionality or service. Additional effort is required in securing the application with respect to increased RPCs. Moving from monolith to microservices, creates comparatively

more complex system architecture as well as complex and challenging work on the testing domain [2], [7].

The positives outweighing the negatives has made the world of software engineering seriously consider microservices in commercial application development.

2.2 Serverless Architecture

Serverless architecture is a relatively new and evolving concept for the world of software engineering. Simply, serverless means that application developers only have to focus on business logic development without focusing on managing and maintaining backend infrastructure [2], [3], [4]. The most important fact to be highlighted is that this does not mean that there are no servers to run the application, instead third-party service providers maintain the entire backend infrastructure on behalf of us. They offer all necessary services such as maintaining servers, load balancing, auto scaling, security, database operations etc. The reference terminology which used for the above implementation is called Backend-as-a-Service (BaaS) or Mobile Backend-as-a-service (MbaaS) [2], [18].

In year 2014, with the introduction of AWS Lambda, Amazon revolutionized the serverless paradigm. The revolutionary change initiated by AWS introducing AWS Lambda functions, had a major impact on the traditional way of backend process running on a server 24/7 anticipating HTTP requests/API calls. AWS built a paradigm where instead of a dedicated server space or a dedicated backend processor running all the time, backend piece of code is executed based on an event triggered mechanism. Considering from a developer's context, he or she can completely ignore focusing on managing and maintaining when being in this paradigm, instead simply focus on writing proper code to be executed upon an event trigger. Cloud service provider will then take up the responsibility of finding a server space to execute the code and manage scaling. This change was noted as Function-as-a-Service (FaaS) [3], [18].

Inception of serverless architectures has made a huge impact on the traditional approach of modeling a system with the basic three-tier architecture. System under the traditional three-tier architecture will probably be consisting of rather an unintelligent client and core business logic written in the server side. Serverless provides the pathway to building applications with rich clients integrated with BaaS and FaaS. In line with the above statement a traditional systems

authentication logic will be in the server side which would be replaced by a BaaS within serverless architecture. Furthermore, a core use case could be replaced by a FaaS based on an event trigger. As mentioned above serverless has been able to add new dimension in the world of software engineering in developing commercialized applications [2], [3].

FaaS the new dimension of serverless architectures, has many characteristic advantages. Fundamentally FaaS provides the advantage of running backend code without managing and maintaining your own servers. FaaS takes the full responsibility in handling the scalability of the application as well as instead of making payments for the total server up time, FaaS introduces a charging mechanism which is based on the usage for its millisecond. FaaS will calculate the server usage based on the incoming requests serving time. In simple terms FaaS reduces both the development and operational costs [3], [18]. In commercial application development one other key benefit that FaaS dimension of serverless architectures provide is the reduction in packaging and deployment complexity. Even though with all the above-mentioned positives there are areas that needs further improvement within serverless architectures. Multitenancy is one major concern of being serverless, where multiple customer applications are running on the same machine. This could lead to security vulnerabilities, concerns on robustness as well as application performance. Vendor lock-in has become another concern as the FaaS implementation becomes vendor specific and raises the concerns of if the developers needs to switch the vendor, then it might be necessary to change operational tools, code and may even need to change application design or architecture [3], [4], [18], [19].

Considering the advantages that FaaS brings into the world of software development has made it a to become a huge impact to the modern-day application architecture. Apart from AWS Lambda, leading tech companies such as Google, Microsoft have entered in facilitating FaaS with the introduction of cloud functions and Microsoft Azure respectively.

2.3 Rule-Based Systems

A rule-based system contains set of rules which is predefined, or which is learned real time and evolves continuously in order to produce outcomes based on the existing rules set. In summary it is special type of expert system [20]. If the rule-based system is modeled to learn from real data, it would have the ability to continuously update its knowledge base. Capturing and refining the human expertise is one of the major capabilities of modern-day rule-based systems [20], [21]. There are many different ways and means which could be used in developing a rule-based system, but all of those commonly share below set of key properties.

- Incorporating human knowledge is done in conditional if-then rules
- Solves complex problems using appropriate rules combining the results in an appropriate manner
- Increment of skill level and the improvement of its knowledge base is directly correlated with each other
- It determines which is the best sequence of rules to be executed

A rule-based system modeled using if-then rules will evaluate the input with the existing condition and then decide the best suited output. As more data comes in, system will learn from the data which flows through it and as a result of that existing rules will be continuously updated. The continuously updating rules will result in a continuous alteration to the output the system provides, improving the overall accuracy of the system. Probabilistic logic, Computational logic, Deterministic logic, Rough logic and Fuzzy logic are some of the front running logic types within rule-based systems [20].

2.4 Domain Specific Software Architecture (DSSA)

Domain-Specific Software Architecture (DSSA) is a collection of software components, which effectively communicate with each other across a specific domain, brought in to a standardized and predefined structure effective for building successful applications [22].

Knowledge on DSSAs, performs highly valuable when previous experience and past architecture knowledge can be brought in to influence on future application development [23], [24], [25], [26]. The key rational behind the above explanation is that if reasonable amount of work has been conducted within a specific domain, that knowledge acquired can lead to a best suitable solution for majority of the applications within that specific domain. Future applications to be developed within a specific domain should always be inspired by the previous obtained knowledge and learnings for the past application development within that domain [22], [25].

2.5 Backend for Frontend (BFF)

BFF defines the concept of containing a dedicated server-side backend for each type of client application, as shown in Figure 4. Fundamentally the dedicated server-side backend is closly coupled to a given client application [27].

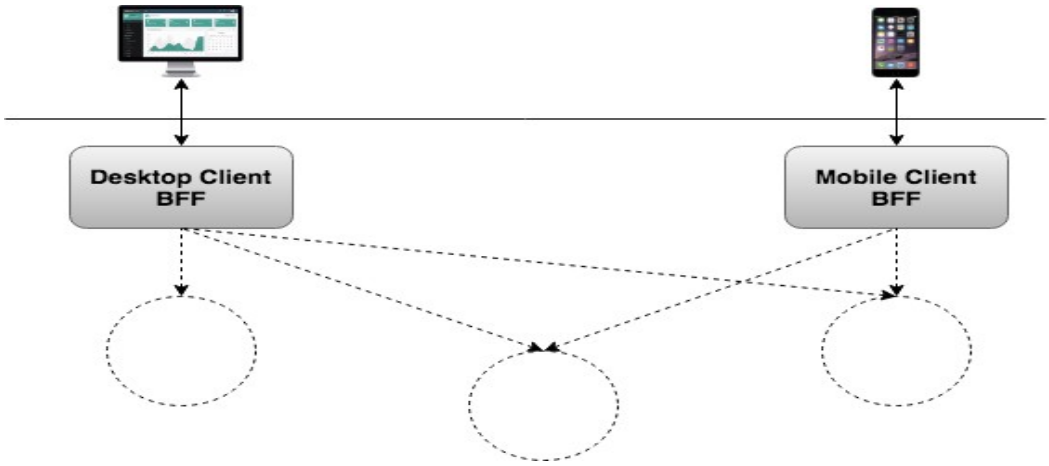

Figure 4: One BFF per user interface

2.6 Service Oriented Architecture (SOA) and Micro-services

Service-oriented architecture (SOA) contains multiple services which collaboratively work with each other in order to facilitate a user need. A service identified in SOA means a completely dedicated process. Communication within each service in a SOA happens across a network rather than method calls.

Microservices approach has emerged from SOA, considering business boundaries and the intended services in order to solve a real-world use case. All in all, microservices could be identified as a specific approach for SOA [7], [28].

2.7 Architecture Description Languages (ADLs) – Model Software Architecture Based Development

ADL for software applications provide a high-level modeling rather than focusing on detail implementation points of source code. Accordingly, ADL must be simple in its own context, understandable, facilitated with tool-based support, visually pleasing and containing graphical syntax to analyze architectural descriptions [12], [14], [31], [32].

Systems high-level architecture is basically brought up of various components and its connectors. In order to ADLs to provide a proper representation of systems architecture it needs to model systems components, connectors and its configurations [31], [32]. Further, it is mandatory for an ADL tool to provide support for architecture-based development and evolution in order to truly become useful. Even though a considerable amount of research has happened in terms of tool-based support for software architecture domain still there is a definite gap to be bridged between the research scope and actual need. A most common fact on ADLs are that though they provide tool-based support they focus on single purpose such as architecture refinement or analysis [12], [14], [29], [30], [31].

Following is an analysis done on several ADLs comparing its effectiveness in modeling components, connectors, architectural configurations and tool support [14].

Table 1: ADL Facilitation for Modeling Components [14]

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Component*; implementation independent	interface points are *ports*	extensible type system; parameterization enabled with templates	no support; can use other ADLs' semantic models in property lists	via interfaces only	none	allows any attribute in property lists, but does not operate on them
Aesop	*Component*; implementation independent	interface points are *input* and *output ports*	extensible type system	(optional) style-specific languages for specifying semantics	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with components
C2	*Component*; implementation independent	entire interface one *port*; interface elements are *messages* (*notifications* and *requests*)	extensible type system	causal relationships between input and output messages	via interfaces and semantics; stylistic invariants	name, interface, behavior and implementation subtyping (and their combinations)	none
Darwin	*Component*; implementation independent;	interface points are *services* (*provided* and *required*)	extensible type system; supports parameterization	π-calculus	via interfaces and semantics	none	none
MetaH	*Process*; implementation constraining	interface points are *ports*	Predefined, enumerated set of types	ControlH for modeling algorithms in the GN&C domain; implementation semantics via paths	via interfaces and semantics; modes; non-functional attributes	none	attributes needed for real-time schedulability, reliability, and security analysis
Rapide	*Interface*; implementation independent	interface points are *constituents* (*provides*, *requires*, *action*, and *service*)	extensible type system; contains a types sublanguage; supports parameterization	partially ordered event sets (posets)	via interfaces and semantics; algebraic constraints on component state; pattern constraints on event posets	inheritance (structural subtyping)	none
SADL	*Component*; implementation independent;	interface points are input and output *ports* (*iports* and *oports*)	extensible type system; component types tightly coupled to styles	none	via interfaces; stylistic invariants	component refinement via pattern maps	none
UniCon	*Component*; implementation constraining	interface points are *players*	predefined, enumerated set of types	event traces in property lists	via interfaces and semantics; attributes; restrictions on players that can be provided by component types	none	attributes for schedulability analysis
Wright	*Component*; implementation independent;	interface points are *ports*; port interaction semantics specified in CSP	extensible type system	not the focus; allowed in CSP	protocols of interaction for each port in CSP	none	none

Table 2: ADL Facilitation for Modeling Connectors [14]

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols	no support; can use other ADLs' semantic models in property lists	via interfaces only	none	allows any attribute in property lists, but does not operate on them
Aesop	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols	(optional) semantics specified in Wright (CSP)	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with connectors
C2	*Connector*; explicit	interface between connector and each component given through a separate *port*; interface elements are *messages*	extensible type system, based on protocols	partial semantics specified via message filters	via semantics; stylistic invariants (each port participates in one link only)	none; current focus is on component evolution	none
Darwin	*Binding*; in-line; no explicit modeling of component interactions	none; allows "connection components"	none	none	none	none	none
MetaH	*Connection*; in-line; allows connections to be optionally named	none	none; supports three general classes of connections: port, event, and equivalence	none	none	none	none
Rapide	*Connection*; in-line; complex reusable connectors only via "connection components"	none; allows "connection components"	none	posets; conditional connections	none	none	none
SADL	*Connector*; explicit	connector signature specify the supported data type	predefined, enumerated set of types, one per style	implicit in connector's type (e.g., dataflow)	via interfaces; stylistic invariants	connector refinement via pattern maps	none
UniCon	*Connector*; explicit	interface points are *roles*	predefined, enumerated set of types	implicit in connector's type; semantic information can be given in property lists	via interfaces; restricts the type of players that can be used in a given role	none	attributes for schedulability analysis
Wright	*Connector*; explicit	interface points are *roles*; role interaction semantics specified in CSP	extensible type system, based on protocols	connector *glue* semantics in CSP	via interfaces and semantics; protocols of interaction for each role in CSP	supports type conformance for behaviorally related protocols	none

Table 3: ADL Facilitation for Modeling Architectural Configurations [14]

Features / *ADL*	Character.	Understand.	Composition.	Heterogen.	Constraints	Refinement/ Traceability	Scalability	Evolution	Dynamism	Non-Funct. Properties
ACME	*Attachments*; explicit	explicit, concise textual specification	provided via templates, representations, and rep-maps	open property lists; required explicit mappings across ADLs	ports may only be attached to roles and vice versa	none	aided by explicit configurations; hampered by fixed number of roles	aided by explicit configurations; no support for application families;	none	none
Aesop	*Configuration*; explicit	explicit, concise graphical specification; parallel type hierarchy for visualization	provided via representations	allows multiple languages for modeling semantics; supports development in C++ only	ports may only be attached to roles and vice versa; programmable stylistic invariants	none	aided by explicit configurations; hampered by fixed number of roles	no support for partial architectures or application families; aided by explicit configurations	none	none
C2	*Architectural Topology*; explicit	explicit, concise textual and graphical specification	allowed; supported via internal component architecture	enabled by internal component architecture; supports development in C++, Java, and Ada	fixed stylistic invariants	none	aided by explicit configurations and variable number of connector ports	allows partial architectures; aided by explicit configurations; no support for application families;	pure dynamism: element insertion, removal, and rewiring	none
Darwin	*Binding*; in-line	implicit textual specification which contains many connector details; provides graphical notation	supported by language's composite component feature	allows multiple languages for modeling semantics of primitive components; supports development in C++ only	provided services may only be bound to required services and vice versa	supports system generation when implementation constraining	hampered by in-line configurations	no support for partial architectures or application families; hampered by in-line configurations;	constrained dynamism: runtime replication of components and conditional configuration	none
MetaH	*Connections*; in-line	implicit textual specification which contains many connector details; provides graphical notation	supported via macros	supports development in Ada only; requires all components to contain a process dispatch loop	*applications* are constrained with non-functional attributes	supports system generation; implementation constraining	hampered by in-line configurations	no support for partial architectures or application families; hampered by in-line configurations;	none	supports attributes such as execution processor and clock period
Rapide	*Connect*; in-line	implicit textual specification which contains many connector details; provides graphical notation	mappings relate an architecture to an interface	supports development in VHDL, C/C++, Ada, and Rapide	refinement maps constrain valid refinements; timed poset constraint language	refinement maps enable comparative simulations of architectures at different levels	hampered by in-line configurations; used in large-scale projects	no support for partial architectures or application families; hampered by in-line configurations;	constrained dynamism: conditional configuration and dynamic event generation	timed poset model allows modeling of timing information in the constraint language
SADL	*Configuration*; explicit	explicit, concise textual specification	allowed in principle; no support	component and connector types are tightly tied to its supported styles	programmable stylistic invariants; refinement maps constrain valid refinements	refinement maps enable correct refinements across styles	aided by explicit configurations; used in large-scale project	no support for partial architectures or application families; aided by explicit configurations;	none	none
UniCon	*Connect*; explicit	explicit textual and graphical specification; configuration description may be distributed	supported through composite components and connectors	supports only predefined component and connector types	players may only be attached to roles and vice versa	supports system generation; implementation constraining	aided by explicit configurations and variable number of connector roles	no support for partial architectures or application families; aided by explicit configurations;	none	none
Wright	*Attachments*; explicit	explicit, concise textual specification	allowed in principle; no support	supports both fine- and coarse-grain elements	ports can only be attached to roles and vice versa	none	aided by explicit configurations; hampered by fixed number of roles; used in large-scale project	suited for partial specification; aided by explicit configurations; no support for application families;	none	none

Table 4: ADL Tool-Based Support [14]

Features / *ADL*	Active Specification	Multiple Views	Analysis	Refinement	Code Generation	Dynamism
ACME	none	textual; "weblets" in *ACME-Web*; animation of pipe-and-filter architectures; architecture views in terms of high-level (template), as well as basic constructs	parser	none	none	none
Aesop	syntax-directed editor for components; visualization classes invoke specialized external editors	textual and graphical; style-specific visualizations; parallel visualization type hierarchy; component and connector types distinguished iconically	parser; style-specific compiler; type checker; cycle checker; checker for resource conflicts and scheduling feasibility	none	*build* tool constructs system glue code in C++ for pipe-and-filter style	none
C2	design critics and to-do lists in *Argo*	textual and graphical; view of development process	parser; critics to establish adherence to style rules and design heuristics	none	class framework enables generation of C/C++, Ada, and Java code	*ArchShell* allows *pure* dynamic manipulation of architectures
Darwin	automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties;	textual, graphical, and hierarchical system view	parser; compiler; "what if" scenarios by instantiating parameters and dynamic components	compiler; primitive components are implemented in a traditional programming language	compiler generates C++ code	compilation and runtime support for *constrained* dynamic change of architectures (replication and conditional configuration)
MetaH	graphical editor requires error correction once architecture changes are *applied* and constrains the choice of component properties via menus	textual and graphical; component types distinguished iconically	parser; compiler; schedulability, reliability, and security analysis	compiler; primitive components are implemented in a traditional programming language	compiler generates Ada code (C code generation planned)	none
Rapide	none	textual and graphical; visualization of execution behavior by animating simulations	parser; compiler; analysis via event filtering and animation; constraint checker to ensure valid mappings	compiler for executable sublanguage; tools to compile and verify event pattern maps during simulation	executable system construction in C/C++, Ada, VHDL, and Rapide	compilation and runtime support for *constrained* dynamic change of architectures (conditional configuration)
SADL	none	textual only	parser; analysis of relative correctness of architectures with respect to a refinement map	checker for adherence of architectures to a manually-proved mapping	none	none
UniCon	graphical editor prevents errors during design by invoking language checker	textual and graphical; component and connector types distinguished iconically	parser; compiler; schedulability analysis	compiler; primitive components are implemented in a traditional programming language	compiler generates C code	none
Wright	none	textual only; model checker provides a textual equivalent of CSP symbols	parser; model checker for type conformance of ports to roles; analysis of individual connectors for deadlock	none	none	none

2.8 Formal Process for Software Architecture Improvement

Having a proper software application architecture is utmost necessary specially for larger, complex and mission critical systems. Few years back the importance given on having a proper software architecture for the developed system was comparatively very low and as the system needed to incorporate changes and move forward it became more difficult to embrace change. To make such changes in the software, it is vital to initially adapt the software to embrace changes which more importantly require architecture improvements to the existing application [7], [9], [33], [34].

The research work carried out within this extreme has produced a methodology which analyses the existing software architecture and evaluates ways to improve it as well as ways to embrace change using Relation Partition Algebra (RPA) model and improves the existing architecture with the best suitable value additions. This complete end to end process is known as the software architecture improvement cycle which is shown in Figure 5 and it is made up of four main steps [7], [9].

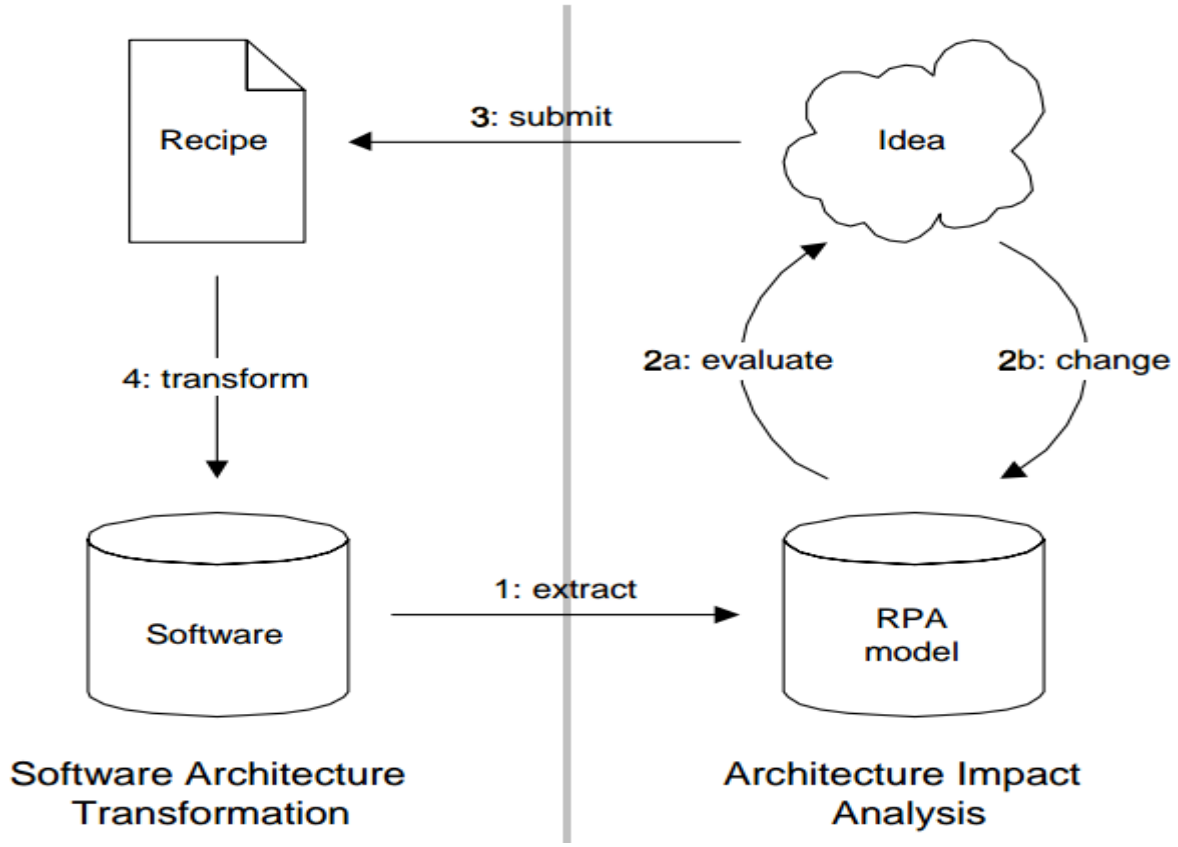

Figure 5: Software Architecture Improvement Cycle [9]

Step 1: Extract

Considering the inputs provided by the architects, the architecture description is extracted from the software. Software architecture description described in this section refers to the relations between so-called design entities. Considered design entities levels of abstraction which leads to functions, models, components etc. RPA model contains the result of extract step.

Step 2a: Evaluate

Here in step 2 the evaluation of the RPA model is conducted. The evaluation will produce an image of the application architecture according to the RPA model. Most importantly structures of the software will be visualized as well as calculation of quality aspects would happen. The benefit of outcomes of this step would be that the architect would be able to provide ideas to change the existing RPA model in order to finetune the quality aspects.

Step 2b: Change

Outcomes of step 2a will lead to the beginning of step 2b. Hence both “Evaluate”, and “Change” are closely connected sub parts of step 2. Identified changes of step 2a will be conducted in step 2b. Importance of this step is that no changes are done to the actual software, rather the changes are done to the abstract model. It is far easier to change the abstract model as well as change results are available quickly. This further allows the architect to try different changes as the changes are only imposed on the abstract model hence the software is not corrupted. The nature of step 2 is after each “Change” step an “Evaluate” step would be executed.

Step 3: Submit

The finalized changes on step 2 is used to submit the recipe. The order list of changes to be performed upon the actual software is known as the recipe. The transformations to carry out in step 4 will be entirely based on the recipe formulated here.

Step 4: Transform

Step 4 will be executed based on the recipe provided by step 3. Automating the transform step will result in eliminating human errors which could occur if done manually as well as speed up the process. Upon completion of step 4 software will again reflect the RPA model. Software architecture improvement process can be restarted without conducting the extract step, assuming the tool chain is error free.

2.9 Scenario-Based Software Architecture Evaluation

Software architecture analysis and evaluation has become a well-established important practice within the architecting paradigm of the software systems. Increased demand to achieve high quality software systems has led it to be so. As the system gets more complex the development effort, time to market, cost goes higher. The architects as well as developers use various tools and methodologies to evaluate the quality of a system against its requirements [35]. After conducting various research initiatives by various different research groups wide variety of methods have been introduced for software architecture quality evaluation [10], [11], [12], [13], [14], [36], [37].

Software Architecture Analysis Method (SAAM)

Industry identifies SAAM as the First mainstream scenario-based software architecture analysis method. One of the most highlighting factors of SAAM is that it has the ability to quickly assess many quality attributes such as modifiability, integrability, extensibility, portability and functional coverage. Further SAAM asses the non-functional quality aspects such as performance and reliability [10], [11], [13].

Architecture Tradeoff Analysis Method (ATAM)

ATAM is an improved version of SAAM to assess the quality attributes such as portability, extensibility, modifiability, and integrability. The main improvement over SAAM is that ATAM not only assess the quality attributes but also its interactions and interdependencies among them too. This assessment allows to highlight opportunities and trade-off mechanisms between each of the identified quality metrics.

In summary ATAM is based on SAAM, but yet an improved version of SAAM which primarily focus on how effectively and efficiently current software architecture satisfies identified quality goals [10], [11], [12].

Cost Benefit Analysis Method (CBAM)

CBAM main focus areas are analyzing benefits, costs as well as implications of architectural decisions. Most importantly CBAM evaluates the level of uncertainty associated with architectural decisions hence it allows the architects to make their decisions based on more informed basis. Another element of CBAM is it acts as a bridge between the architecting process and the economics of the software development organization.

Preliminary focus of SAAM and ATAM was around the quality attributes such as performance, availability, usability, modifiability etc. CBAM makes a mark on claiming architectural costs, benefits and risks are highly important and should be given the same consideration compared to the quality attributes focused by SAAM and ATAM [10], [11], [14].

Architecture Level Modifiability Analysis (ALMA)

ALMA is another type of scenario-based analysis method which focus on software architecture modifiability. It allows to conduct software architecture modifiability assessment by having set of indicators, assessing the risk and predicting maintenance cost. ALMA supports comparing different systems among each other supporting software architecture selection as well. In order to conduct multi system comparison ALMA uses change-scenarios. As the first step of the modifiability assessment, a set of scenarios are identified which might happen during the evolution of the system and then evaluate how well the current architecture embrace change [10], [15], [16].

Family – Architecture Analysis Method (FAAM)

FAAM is a special type of scenario-based software architecture assessment technique conducted upon information systems, focusing on two aligning quality aspects, extensibility and interoperability. The main purpose of FAAM is to create a process to evaluate and assess information system family architectures [10], [17].

3. Methodology

The modularized architecture of TheArchitect, is illustrated in Figure 6. In order to explain each modules workflow, I will be using a real-world example project (Commission Calculator) which is designed to calculate commission for the sales agents.

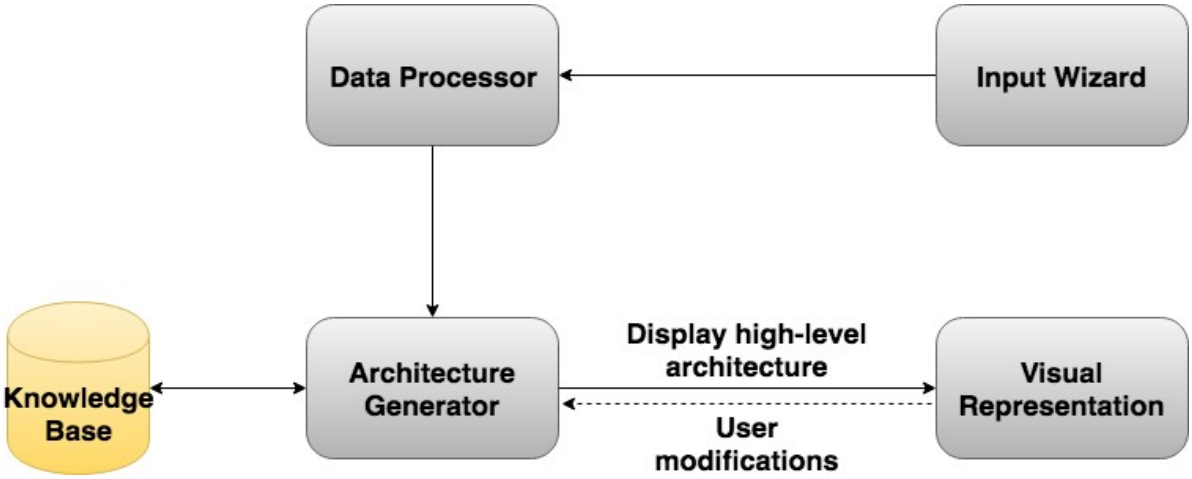

Figure 6: High-level architecture of TheArchitect

3.1 Input Wizard

The input wizard initially requests to specify the focused application domain for the considered application from a list of application domains (e.g., finance is the application domain for Commission Calculator system). If the focused application domain does not appear within the provided list of applications, user has the option to add it to the list. Next, the set of system requirements need to be provided to the system.

Following are the information which will be fed to TheArchitect via the Input wizard component.

- API availability – Contains information about the necessary APIs (e.g., In order to get financial details Commission Calculator desktop application needs to consume finance API).

- Data Storage – Contains the necessity of a database.

- Read / Write Data privileges – Contains information on the responsibility of each use case to write/read data to/from database.

- Client application/s – Contains information on which client application/s consumes which service/s (e.g., finance services are only related to desktop client application while commission grid related services will be used by both mobile and desktop clients).

Furthermore, the set of rules which will be used within the architecture generator would be determined based on the application domain that the user specified at the beginning of the architecture generation process. In order to follow the concepts of DSSAs [24], TheArchitect maintains a different rule set for each application domain. To start with any domain will be using a base rule set as shown in Figure 7 to generate architecture design specific to that domain until an experience software architect conduct modification upon the system generated architecture which proves better than the system generated architecture.


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


Figure 7: TheArchitect - Base rule set for any application domain

3.2 Data Processor

Responsibility within the Data processor is to map the system requirement related information into set of predefined models. TheArchitect becomes a domain independent solution with its ability to map system requirements of any domain to a predefined set of models. Internal models of TheArchitect are as follows.

- Application model – Variations of client applications the system contains are collected within the Application model (e.g., Commission Calculator system requires a mobile and desktop client).

- Service model – Service model will contain details about each system requirement, respective feature category and the respective client application/s which consumes it (e.g., Obtaining and modifying financial statistics for the system are captured under finance feature which only has access via desktop application).

- Data store model – Data store model contains API information which requires for functionality of the system as well as data storage information (read / write capabilities).

3.3 Architecture Generator

Architecture generator determines the relevant client-side applications, BFFs, serverless-microservices, data sources and the communication within each other. High-level architecture generation algorithm (Algorithm 1) will start processing the models received from data processor module.

Algorithm 1 (Figure 11, Figure 12, Figure 13) replicates the way that an experienced architect thinks through in designing a system modeled with serverless-microservices. Next, the processed set of components will then be flown through the flow shown in Figure 16. This will verify and ensure that the finalize components and interactions among them are adhering to the domain specific rule set. The only possible way where the domain specific rule set to be updated would be based on the acceptance of a suggested change on the system generated architecture by an experienced architect. If the rules within the relevant application domain has

not been modified in a previous usage of TheArchitect, it will contain the initial state as represented in Figure 7.

3.4 Visual Representation

Responsibility of this module is to display the generated serverless-microservices by TheArchitect. Figure 8 contains components and interactions among them. The serverless technology analysis is listed in Figure 9. Furthermore, visual representation module allows the users to interact with the system in order to suggest modifications on the system generated architecture.

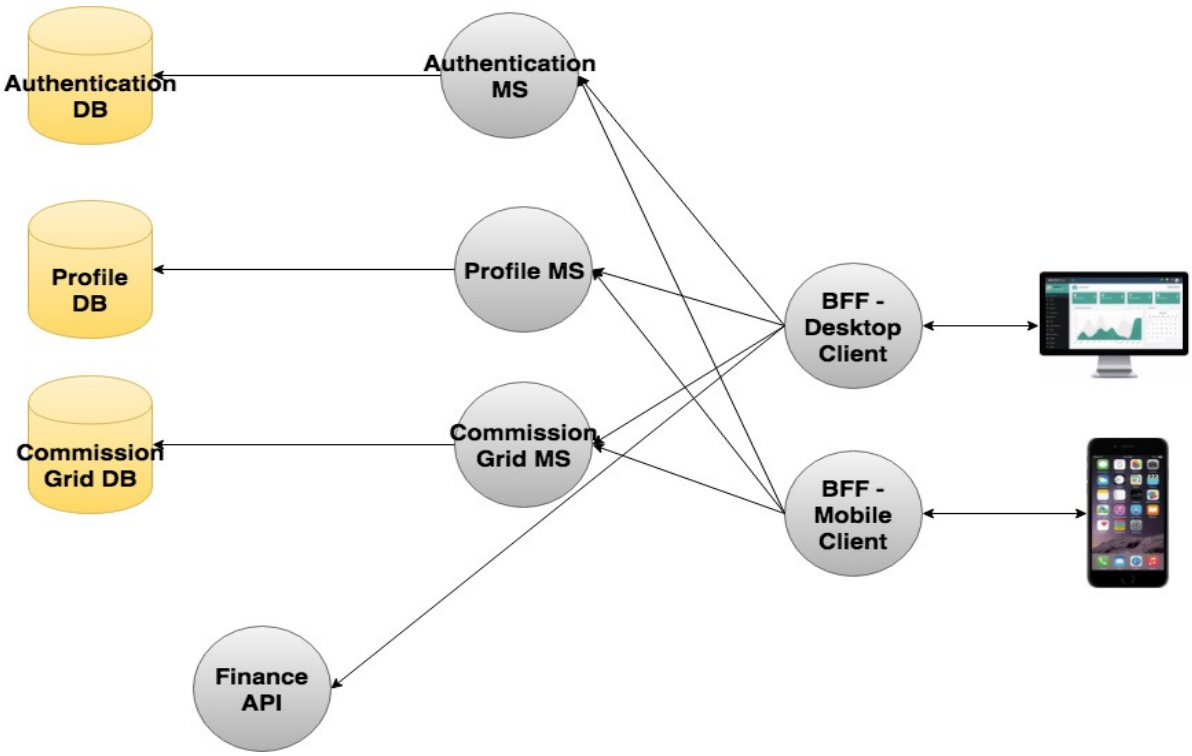

Figure 8: Commission Calculator Application - High-level architecture design diagram

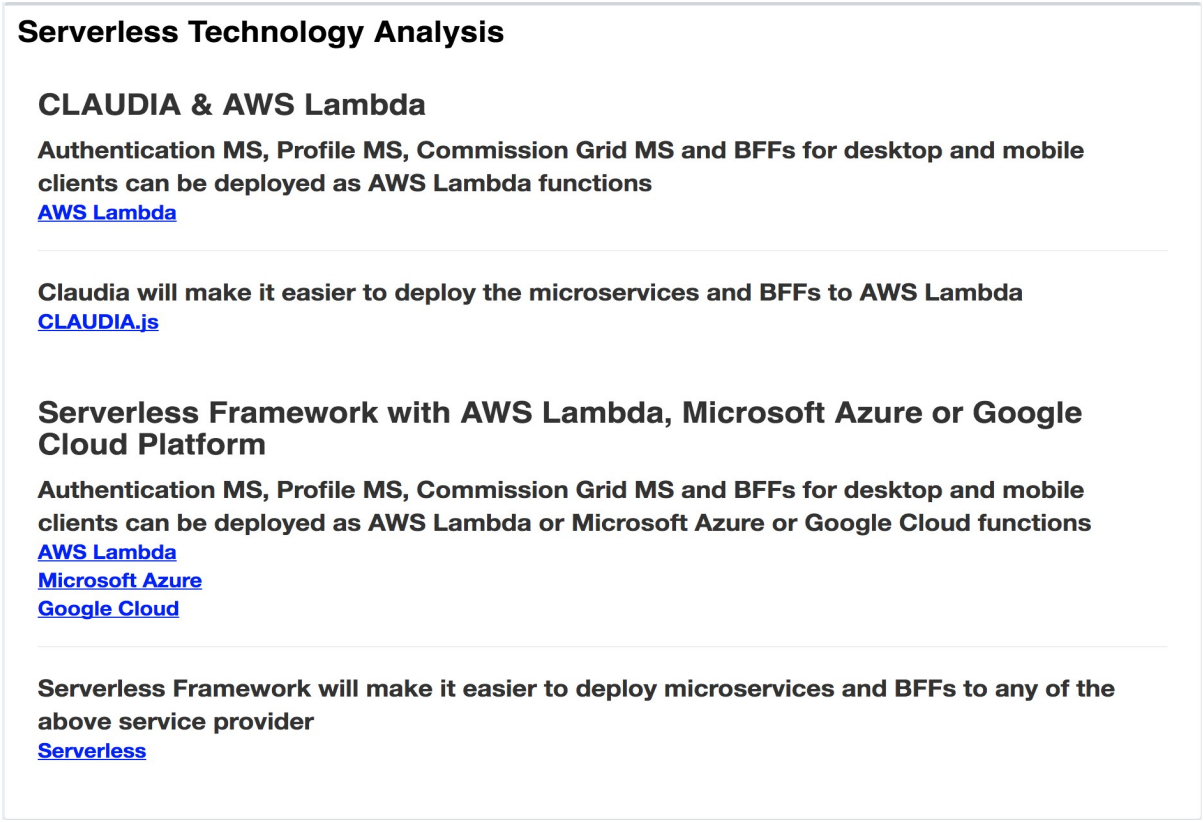

Figure 9: Commission Calculator - Serverless technology analysis

4. Implementation

The following section contains the implementation details regarding the knowledge base, architecture generation algorithm and visual representation module. These have been already published as my research contributions in [38] and [39].

4.1 Knowledge Base

Knowledge base comprises of set of documents each containing a different set of rules associated with a respective application domain. Figure 7 contains the base rules set which is applied to any application domain. Once the user modifications are accepted, the base rule set will be modified (e.g., If a system generated architecture for Commission Calculator system is modified and accepted, it will result modifying the base rules set under the respective application domain as shown in Figure 10).

Furthermore, user modifications will be only accepted if the weighted average score of the metrics for modified software architecture surpasses the system generated architecture weighted average score. The considered metrics are listed under metrics-based evaluation [7] technique.

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

Figure 10: Commission Calculator - Updated rules set for finance domain

4.2 Architecture Generation Algorithm

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
```

Figure 11: Algorithm 1 – High-level architecture generation algorithm

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 applica-
   tion 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
```

Figure 12: Algorithm 2 – Path builder algorithm

Algorithm 3 Clear Path Algorithm

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

Figure 13: Algorithm 3 – Clear path algorithm

Figure 11 contains the algorithm for high-level architecture generation along with Figure 12 and 13 denoting the path builder and clear path algorithms which is used within Algorithm 1 (Figure 11). Set of services and dataStores lists are input parameters for Algorithm 1 (Figure 11). These two params respectively contains service models and database models. One model object representing each functional requirement contains within these lists.

Algorithm 1 (Figure 11) initially determines the necessary architectural components for the intended system. Referring to lines 10-14, apps, type, APIs, dbReadStatus and dbWriteStatus contains information on each system requirement with respect to associated client application names, feature category it belongs to, APIs, data store information and data read/write privileges.

To start with, algorithm will check whether the considered requirement is using any external APIs (line 15). If it so next it will check whether there has been a record added to the apiRecords list previously for the same type of a requirement and if it does not find any, a new entry will be added to apiRecords (line 17). In similar terms line 18 checks whether dbRecords list contains an entry related to the type the requirement belongs to and if it founds such a record the old path will be cleared off (line 19) and a new with will be added (line 20). This path modification ensures that the identified components are in line with the fundamentals of microservices (e.g., persisting and fetching profile information both needs to be served by the profile microservice). If it does not find the focused type within dbRecords, then it will check whether it will exist within recordsWithoutDbs list (line 22) and if so, older path is cleared off. Further the new path will be added (line 25).

Next, as shown in line 29 it checks whether the considered system requirement has any associated data base read/write privileges. If at least one such privilege is found related to the focused system requirement, algorithm will evaluate whether a old record exists within the dbRecords list containing the same type (line 30). If no entry found, then type will be added to the dbRecords list (line 31) and then it would check whether there is a record existing under the focused type within apiRecords list (line 32) and if so the old path will be cleared (line 33) and a new path will be added (line 35).

As checked in line 29 if the type related to the focused functional requirement does not have any database privileges, then it would check whether there are no entries in dbRecords (line

38). If it turns out to be true, then an entry will be added to the recordsWithoutDbs list (line 39) and also will be checked whether an entry exists in apiRecords list too (line 40). If it is so the old path will be cleared off (line 41) and a new path will be added (line 43).

All system requirements will be processed through the focused loop from line 7 to line 46, and as the final result of Algorithm 1 (Figure 11), a paths list will be returned. This list will contain all the interactions among the identified serverless-microservices. The Algorithm 1 (Figure 11) which is in the pseudocode format, is simplified and explained using Figure 14 and Figure 15. Figure 14 covers from line 15 – line 28 whereas Figure 15 covers from line 29 – line 45.

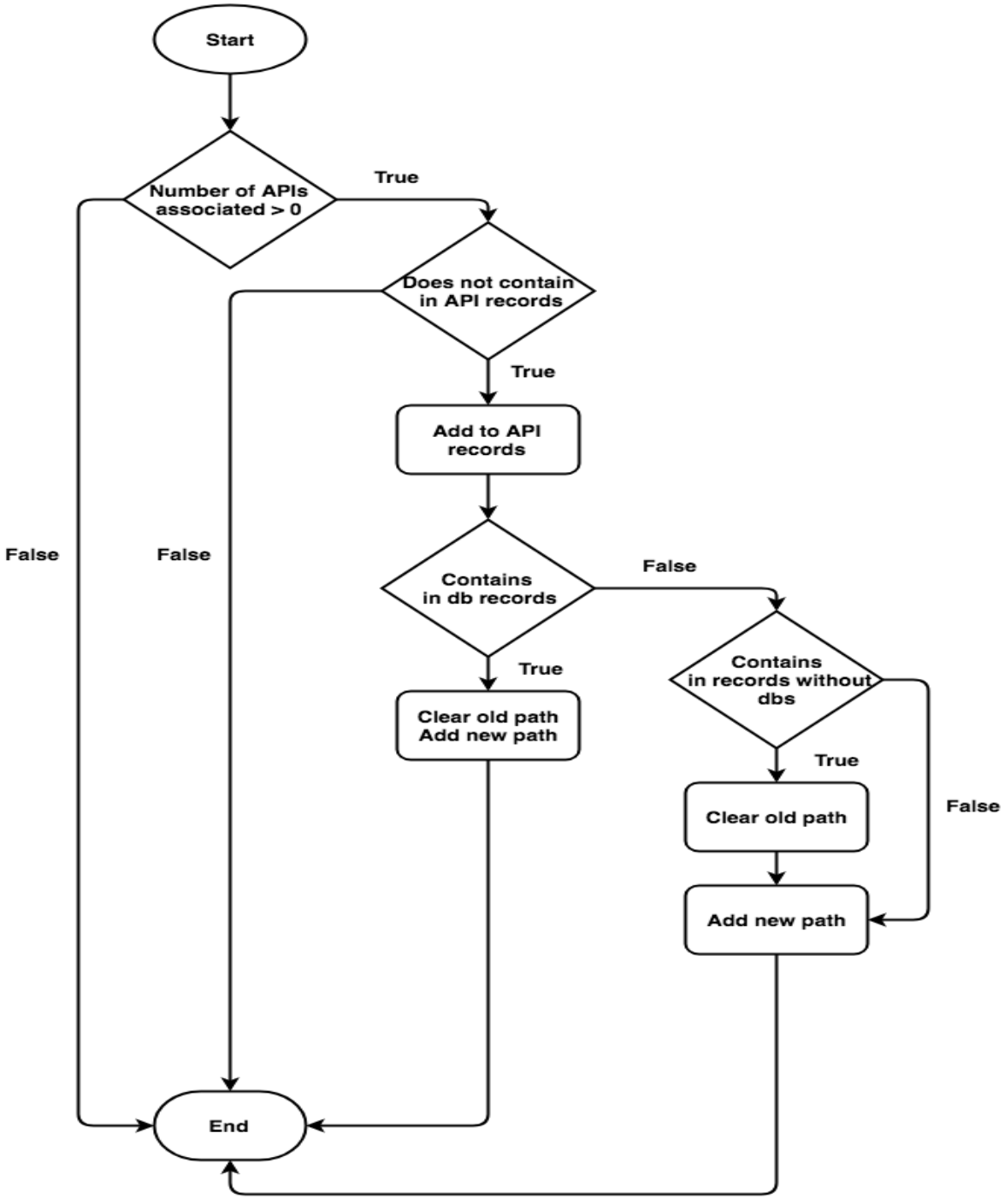

Figure 14: Flow chart – High-level architecture generation algorithm based on API availability

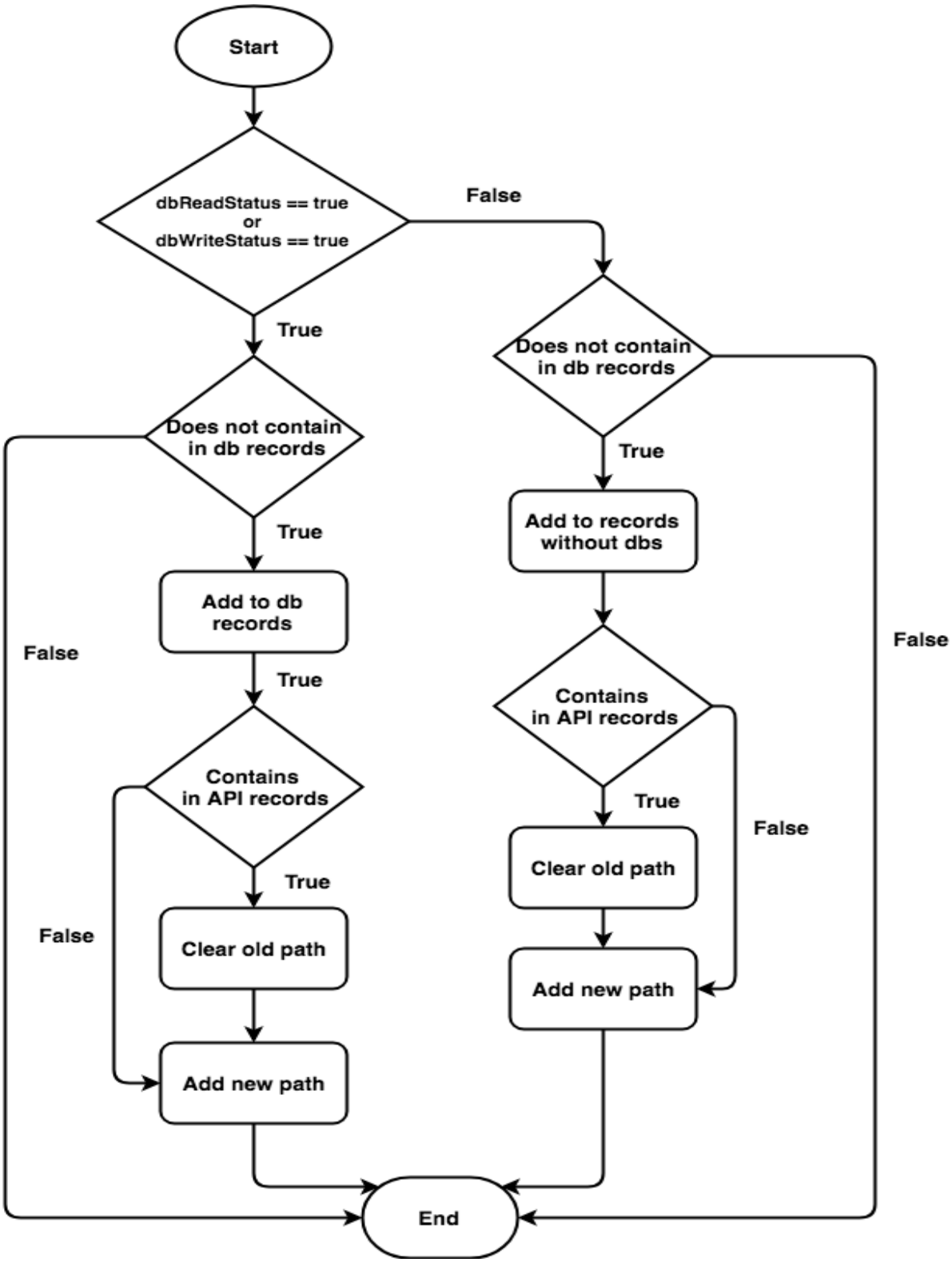

Figure 15: Flow chart - High-level architecture generation algorithm based on database availability

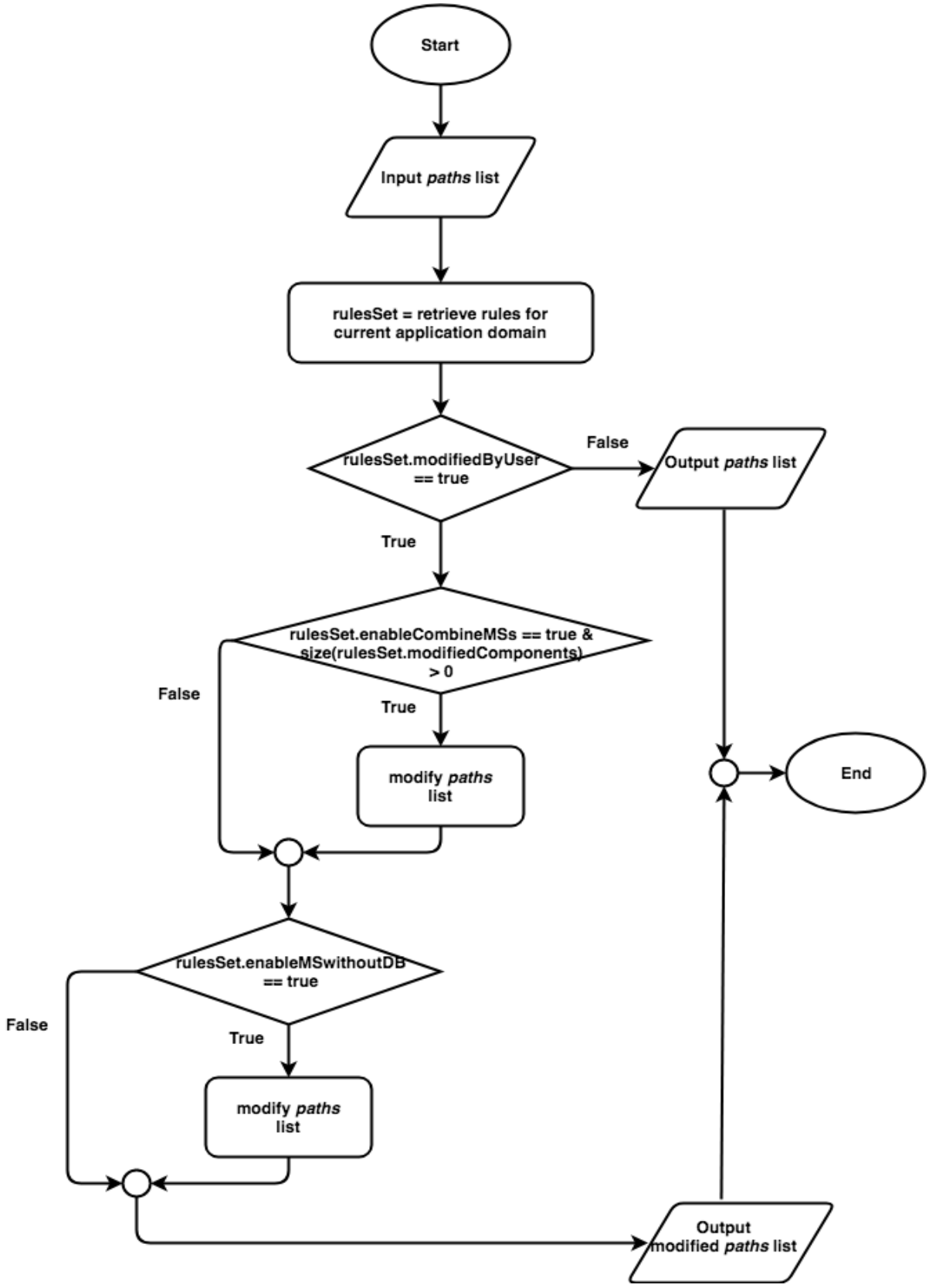

Figure 16: Flow chart - Incorporating domain specific rules set in generating high-level architecture

Next the paths list will be flown through the flow shown in Figure 16. The process will obtain the latest set of rules related to the focused domain and would check whether any user has modified the base rules set by evaluating modifiedByUser field. If a prior modification has not happened means the paths list will not change hence this process stops and visual representation component will be called upon to display the architecture design. Only if a previous modification has taken place it will check upon the enableCombineMSs flag status in order to determine whether there are any components to be combined. In case enableCombineMSs results to be true TheArchitect will check whether there are any components containing a lesser number of connections than which is specified under maxInteractions and if there are any those will be combined with any of the previously combined components. Next, based on the value of the enableMSwithoutDB flag TheArchitect would decide whether to introduce a microservice to a flow which currently does not have a microservice with the intention to decouple the API/DB layer and BFF component/s.

Visual representation model takes the necessary responsibility of focusing on the enableOneBFFperClientApp flag. If the flag is true it will be move ahead with having one BFF per each client application and it is false it will have one BFF for all listed client applications. Finally, after the above modifications the modified paths list will be passed to the visual representation component in order to display the determined high-level architecture.

4.3 Visual Representation

Visual representation component displays the system generated architecture which contains identified set of components as well connections between them (Figure 8). In order to draw the architecture, TheArchitect will iterate via each path element within the paths list. One path representing each feature category is included in the paths list.

Firstly, to formulate the links between BFF components and feature microservices, TheArchitect will figure out the number of client applications and the features belonging to each other. Any feature belonging to a specific client application will have a link to the specific BFF which contains a direct link with its client application. Next, based on the microservices and database statuses the other necessary elements will be drawn. Further, the external APIs, would be added to the design along with the feature links.

The serverless technology analysis will be a static analysis conducted upon the identified microservices as shown in Figure 9. As mentioned previously this module also facilitates in obtaining change requests from experienced architects. Architect is provided with the facility to mark the component that he intends to change and then to specify the other two components that should become the successor and predecessor of the changed component.

5. Experiments

Experimental setup of three different experiments that were conducted to study different aspects of TheArchitect, is presented within this section.

The first type of experiment targeted comparing the system generated serverless-microservices based architecture against an experienced architects design.

Focus of the second type of experiment was to measure the performance/processing time it takes TheArchitect to generate a high-level architecture design containing serverless-microservices.

The focus of the third type of experiment was to evaluate the user preference to use TheArchitect against following the normal process of designing high-level architecture diagrams for custom applications.

5.1 Experiment Design

To evaluate TheArchitect, we used to design architecture designs for multiple enterprise applications. The first type of experiment targeted evaluating TheArchitect which was done using a metrics-based evaluation to generate an overall score of the design architecture. The evaluation parameters and the weights of those parameters were obtained from a panel of industry/domain expert software architects.

Since the tool focused domain driven design, we specifically limited for one domain (restaurant industry) in obtaining real world applications for which the tool generated architecture designs. Further once TheArchitect generates the high-level architecture it was provided to the same panel of architects and allowed them to conduct modifications and checked the evaluation score on the modified architecture and then allowed TheArchitect to update the rule set of architecture generation for the specific domain if the modified architecture exceeded the evaluation score obtained by TheArchitect for the initial design. Further since the experiment was limited to one domain of problems the refined rule set after generating a high-level design was used in designing the next high-level design for the next real-world application. The continuous learning model of TheArchitect allowed it to improve upon each architecture deign

generation as well as reduce the number of problem elements/ manual modifications each time it generates an architecture for a new real-world application in a domain which it had already generated at least one architecture.

Focus of the second type of application was performance/processing times of TheArchitect and in order to evaluate it multiple magnitude real world application system requirements were fed to TheArchitect and measured time to generate a high-level architecture design which approves by the panel of architects.

Third type of experiment focused on the usability evaluation of TheArchitect and in order to obtain statistics on industry likeness to use TheArchitect deviating the traditional process of an experienced architect generating the architecture from scratch each time was evaluated through a questioner-based user study.

5.2 Experiment Type A

The quality of the generated architecture depends on the rule set that used to generate it, which is ultimately specific to the application domain. Here we used multiple similar projects which belongs to the same application domain, with the objective of evaluating the architecture diagrams generated from TheArchitect on even grounds. Following are the metrics considered for the architecture evaluation process [10].

- Coupling – Level of dependency between two components.
  Coupling = Number of components called for each service / Number of services

- Cohesion – Similarity of the facilitated use cases by each component.

- Number of services provided by a component – This determines the distribution of functionality over the entire design.

- Fan in – Number of called services of a component

- Fan out – Number of service calls of a component

- Depth of scenario – Provides a measurement of the level of complexity of a given scenario based on the designed architecture.
  Depth of scenario =
  Total of number of Component interactions for each service / Number of services

If a specific element within the system generated architecture happens to produce an outlying metric value, then that specific element is detected as a problematic element. The values differ more than the twice the standard deviation from the mean value are identified as outlaying values.

As explained above, TheArchitect provides the facility to a software architect to request modifications against the system generated architecture. In order to evaluate the system generated architecture against the experienced architect modified system we use weighted average score of the above explained metrics. These weights would be deciding which metrics are more critical in evaluating the more superior architectural design. Similarly, as the rule set, weights would also be specific to a focused domain. Hence before using TheArchitect to generate architectural designs for a new application domain expert architects need to define the weights which are most appropriate for that domain. If the weights are not predefined TheArchitect will be using equal weights in determining superior design. Furthermore, these weights will only be allowed to tune once to avoid distorting the evaluation process. If this restriction does not take place, then the architect/s could always twist around the weights and force manually generated architecture to be superior than the system generated architecture. Since the objective is to conduct domain driven design, the type A experiment was focused to restaurant management industry related application. Three architects were involved in deciding weights for the metrics which will be used for the architecture evaluations within the restaurant domain. The architects who involved in this process had more than 8 years of experience in designing applications for restaurant management domain. The weights that the panel of architects determined for each metric for the restaurant management domain are shown in Table 5.

Table 5: Weights of evaluation metrics – Restaurant management domain

Metric	Weight
Coupling	0.05
Cohesion	0.05
Number of services provided by a component	0.3
Fan in	0.2
Fan out	0.2
Depth of scenario	0.2

First real-world system that TheArchitect was used to design a high-level architecture design was “Order Receive Application”. “Order Receive” is a mobile application developed for restaurant owners to obtain insights on their incoming deliveries. The mobile application provides real time updates of the delivery time predicting any early/delay arrivals. Further, it also provides information related to invoices associated with each delivery and maintains profile-based preferences to subscribe/unsubscribe for push notifications. Hence this was the first application used to design a high-level architecture, TheArchitect used its base rule set in generating the architecture design and accepted domain experienced architect’s suggestions upon the generated architecture where TheArchitect updated the rule set and created a domain specific rule set which it used in the second experiment in terms of designing architectural design for the second real world application in the restaurant management domain. “Order Receive Application” contained 138 requirements categorized in to 11 high-level system epics.

Second real-world system which TheArchitect generated a high-level architecture design was “Inventory Management Application”. “Inventory Management Application” mainly facilitates in managing inventory within the restaurants. Application contains a desktop and mobile client both. Restaurant owner can place orders, record sales and purchases using the desktop application. Mobile client is developed for the suppliers, from which they can check on orders, accept or reject order requests. Profile based preferences are configurable in both clients. “Inventory Management Application” contained 151 requirements categorized in to 14 high-level system epics.

5.3 Experiment Type B

The performance of TheArchitect was measured based on the processing times it takes to process all the input system requirements and generate the high-level architecture designs.

5.4 Experiment Type C

In terms of experimenting the usability of TheArchitect, we conducted a survey among software architects and software developers to mark whether they like or dislike to use TheArchitect in their day to day work.

6. Results and Discussion

6.1 Order Receive Application

The initially generated architecture diagram by TheArchitect for the Order Receive mobile application is shown in Figure 17 and the architecture diagram with accepted user modifications for the same system is shown in Figure 18. Metrics based evaluation is conducted upon 6 identified components, facilitating 138 system requirements categorized into 11 high-level system epics.

Figure 19 contains the component-based evaluation conducted upon initial system generated architecture along with Figure 20 contains the component-based evaluation conducted upon the architecture with accepted user modifications. Vertical bars in Figure 19 and Figure 20 represents Component based metric values shown for systems architecture without and with user modifications. Horizontal bar in the same two figures shows the anomaly detection benchmark for each metric using the summation of mean and two times the standard deviation value.

Table 6 and Table 7 respectively contains the services-based metrics evaluation for initial system generated architecture and the architecture with accepted user modifications.

In analyzing system generated architecture in Figure 17 against the architecture with accepted user modifications shown in Figure 18 it is clear that user has requested to introduce an *Order MS* from which the *invoices* and *deliveries* will be fetched/updated along with *authentication API* to be invoked via *Profile MS*. Accepting user modifications resulted in eliminating external API invocations directly from BFF and also reduce the number of components focused under the component based metrics evaluation. Even though the deliveries component was a problem element in terms of fan out measure within the system generated architecture, no problem elements found within the amended architecture design. Further the possibility of a component becoming a problem element even in future was also being reduced as a result of the user modifications.

Table 6 figures compared against Table 7 figures proves that the accepted user modifications have negatively impacted upon the coupling and depth of scenario measurements within the

architecture. Yet TheArchitect accepts the requested user modifications relying on the weights provided by the panel of experienced architects which denotes that positive impact of changes outweigh the negative impact created by the two-metrics considered under services-based metrics evaluation.

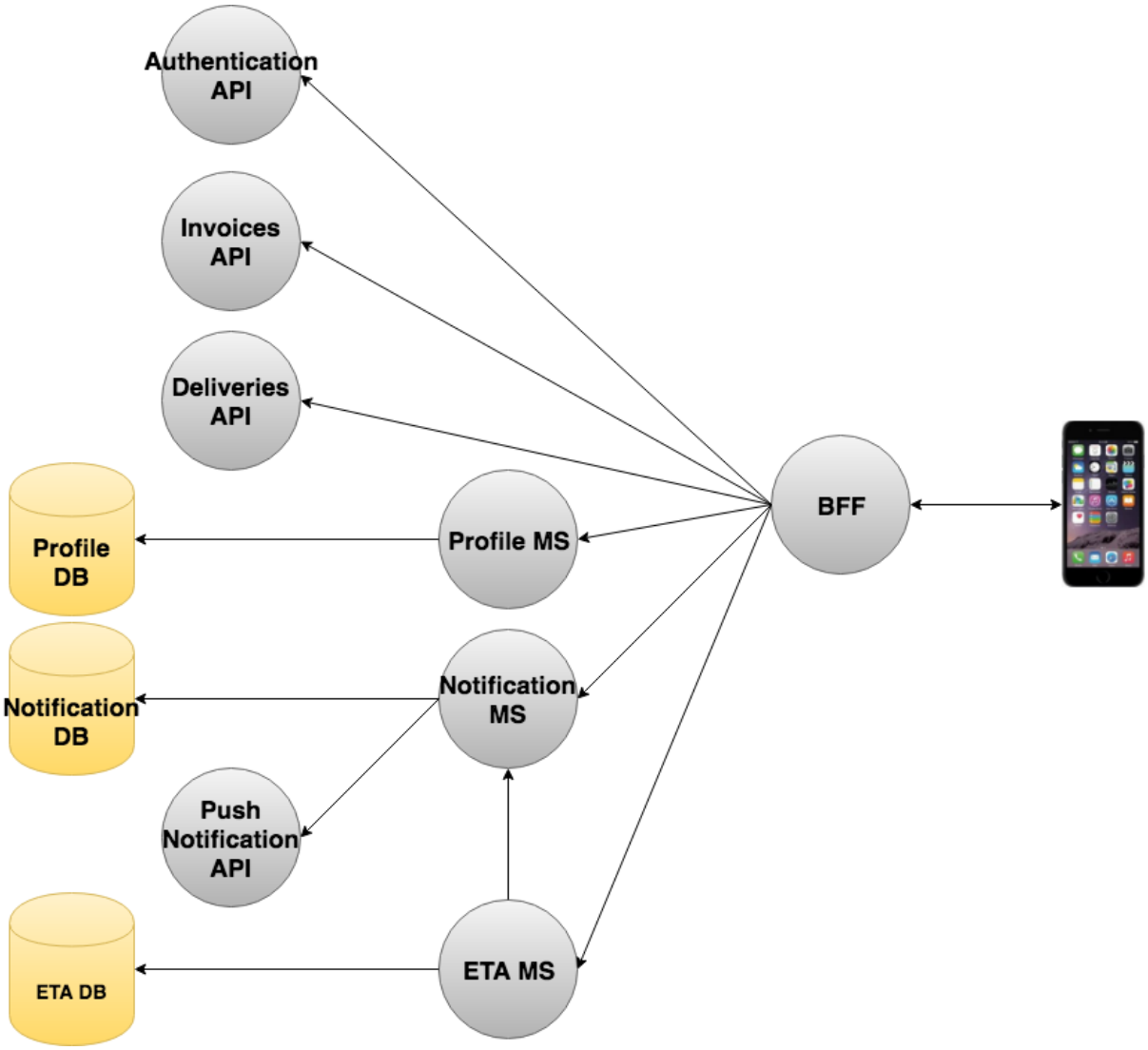

Figure 17: Order Receive Application - System generated high-level architecture design diagram

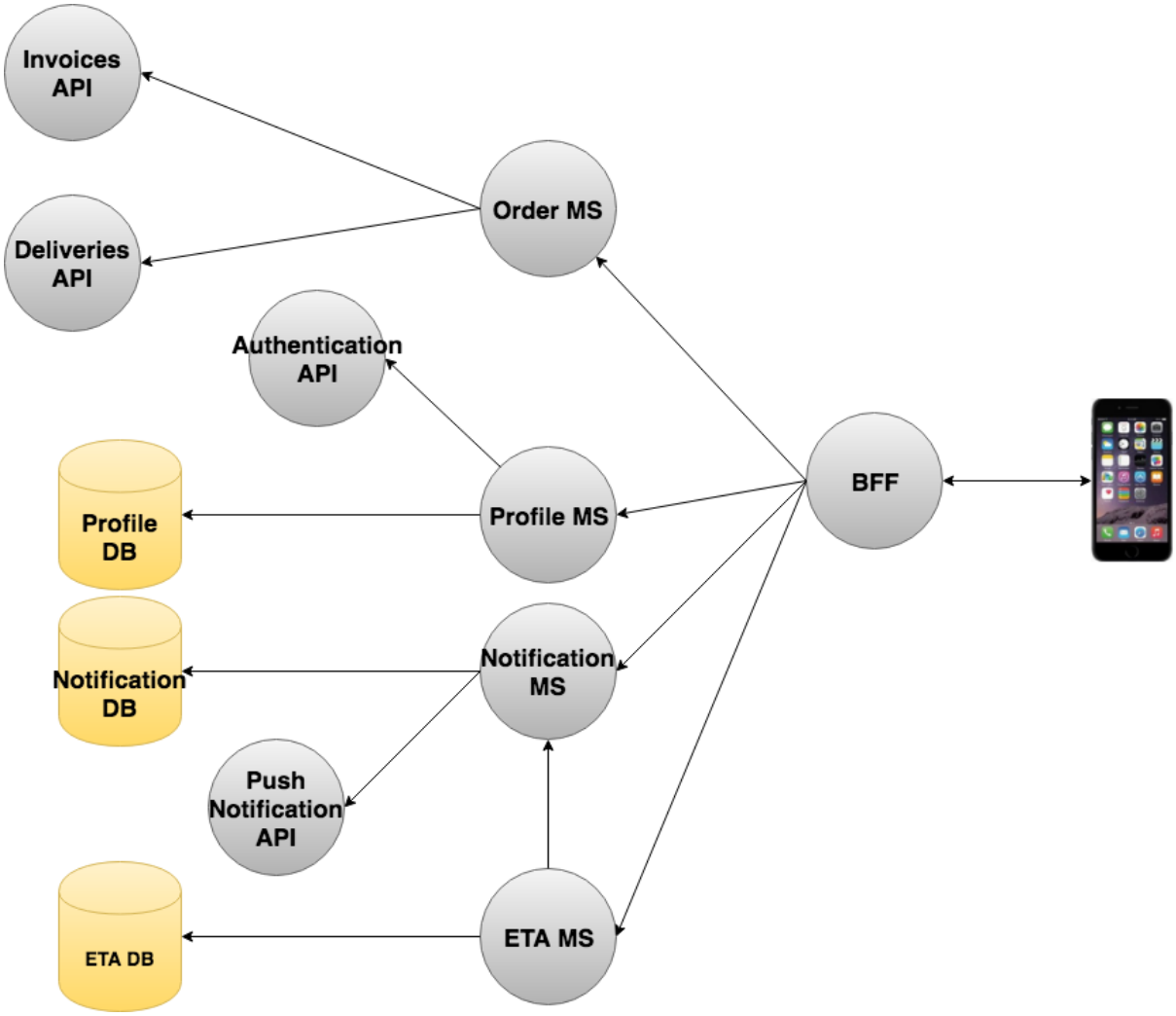

Figure 18: Order Receive Application - High-level architecture design diagram with accepted user modifications

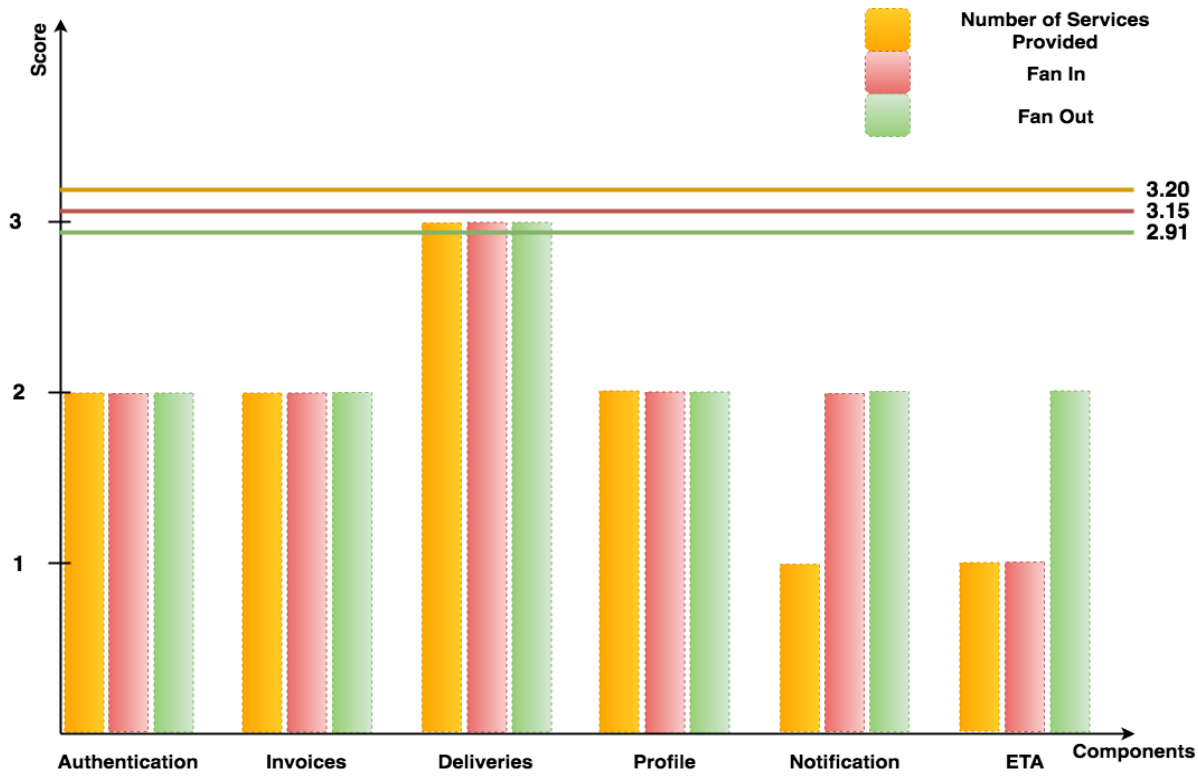

Figure 19: Order Receive Application - Component based metrics evaluation on system generated architecture diagram

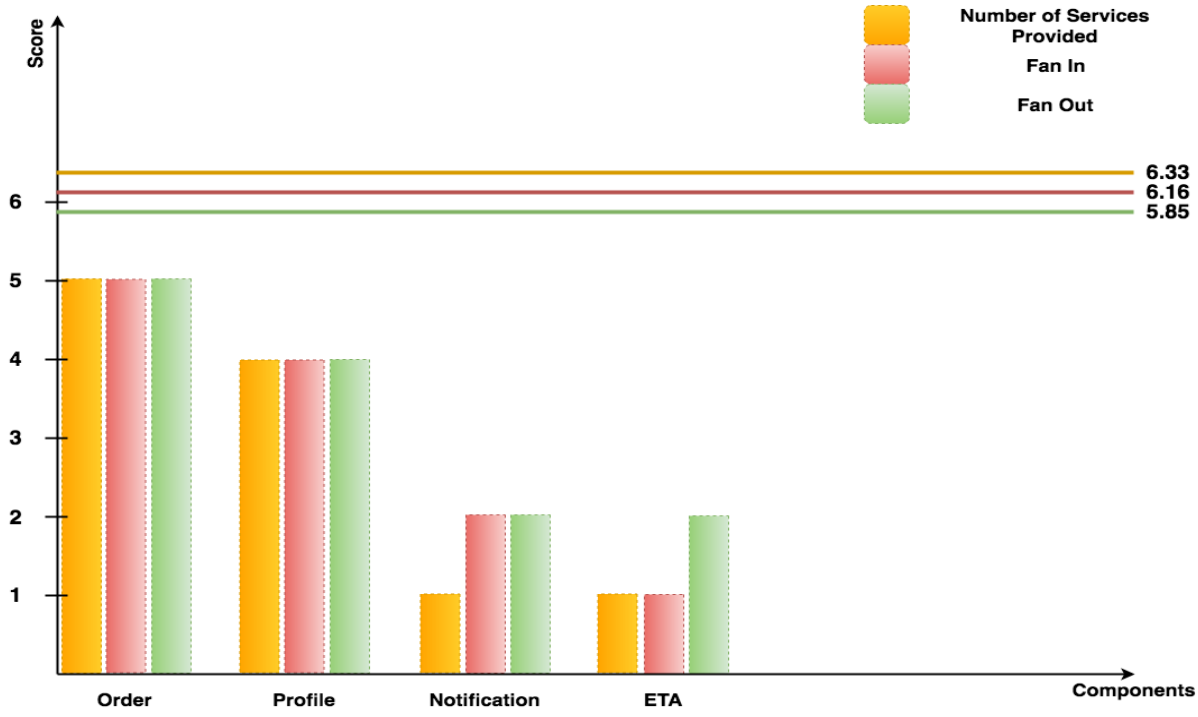

Figure 20: Order Receive Application - Component based metrics evaluation on user modifications accepted architecture diagram

Table 6: Order Receive Application - Services based metrics evaluation on system generated architecture

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

Table 7: Order Receive Application - Services based metrics evaluation on user modifications accepted architecture

Metric	Value
Coupling	1.91 component calls per service
Depth of scenario	2.27 component interactions per service

6.2 Inventory Management Application

Figure 21 shows the initially generated high-level architecture for the Inventory Management application by TheArchitect and the architecture diagram with accepted user modifications for the same system is shown in Figure 22. Metrics based evaluation is conducted upon 4 identified components, facilitating 151 system requirements categorizing into 14 high-level system epics.

Figure 23 contains the component-based evaluation conducted upon initial system generated architecture along with Figure 24 contains the component-based evaluation conducted upon the architecture with accepted user modifications. Vertical bars in Figure 23 and Figure 24 represents Component based metric values shown for systems architecture without and with user modifications. Horizontal bar in the same two figures shows the anomaly detection benchmark for each metric using the summation of mean and two times the standard deviation value.

Table 8 and Table 9 respectively contains the services-based metrics evaluation for initial system generated architecture and the architecture with accepted user modifications.

In analyzing system generated architecture in Figure 21 against the architecture with accepted user modifications shown in Figure 22 it is clear that user has requested to eliminate *purchases microservice* and incorporate all purchases related functionality within the *order microservice* because ultimately purchases are a part of the order. Furthermore, as a result of learning under

took with the architecture generation for Order Receive application, the *authentication API* is being invoked via *Profile MS*. Accepting user modifications resulted in reducing the number of components focused under the component-based metrics evaluation. Even though the system generated architecture did not consist of any problem elements, as a result of accepting user's modifications, the possibility of a component becoming a problem element even in future has also being reduced.

Table 8 figures compared against Table 9 figures proves that the accepted user modifications have result in 0.01 reduction in the coupling measurement and no change in the depth of scenario measurement. TheArchitect accepts the requested user modifications relying on the weights provided by the panel of experienced architects since user requested modifications has improved both component and services based metric evaluation statistics.

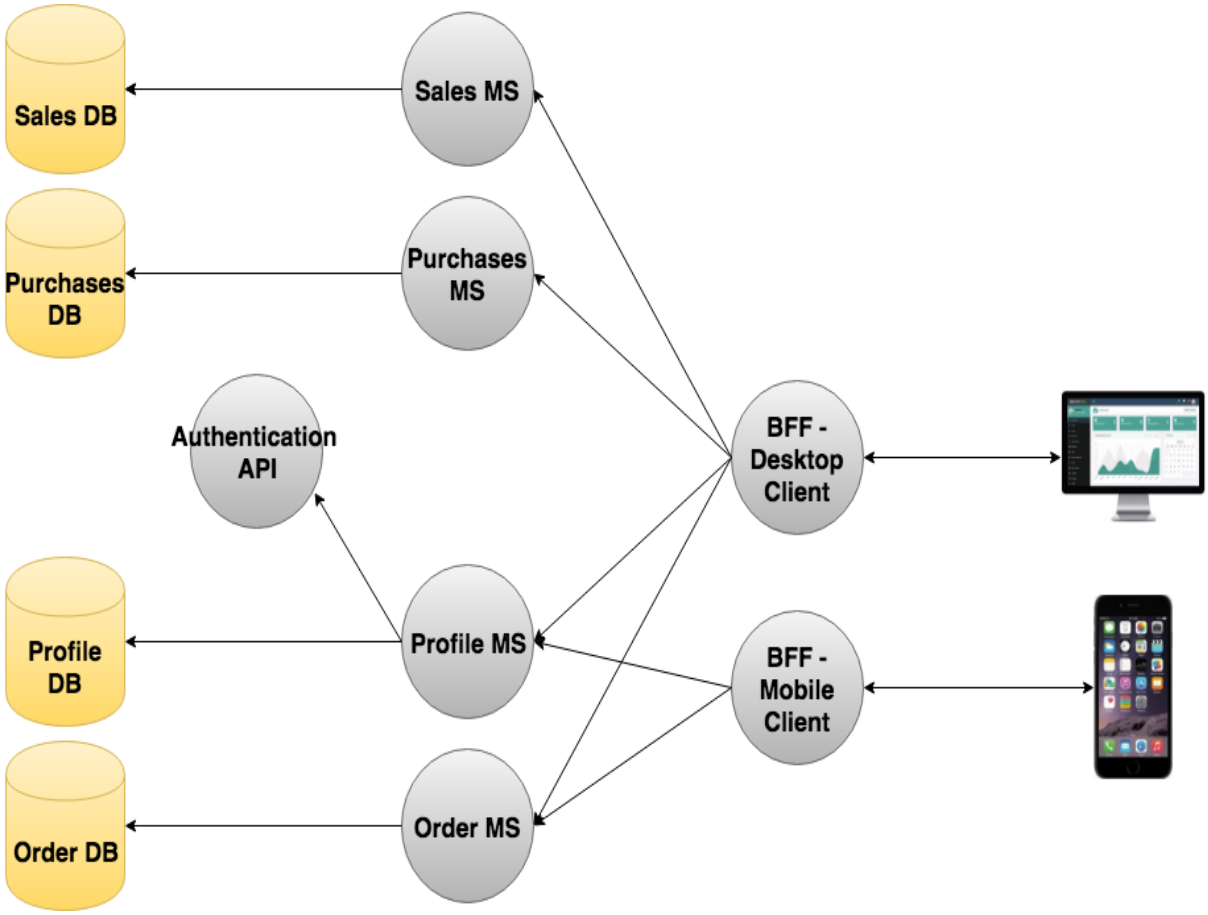

Figure 21: Inventory Management Application - System generated high-level architecture design diagram

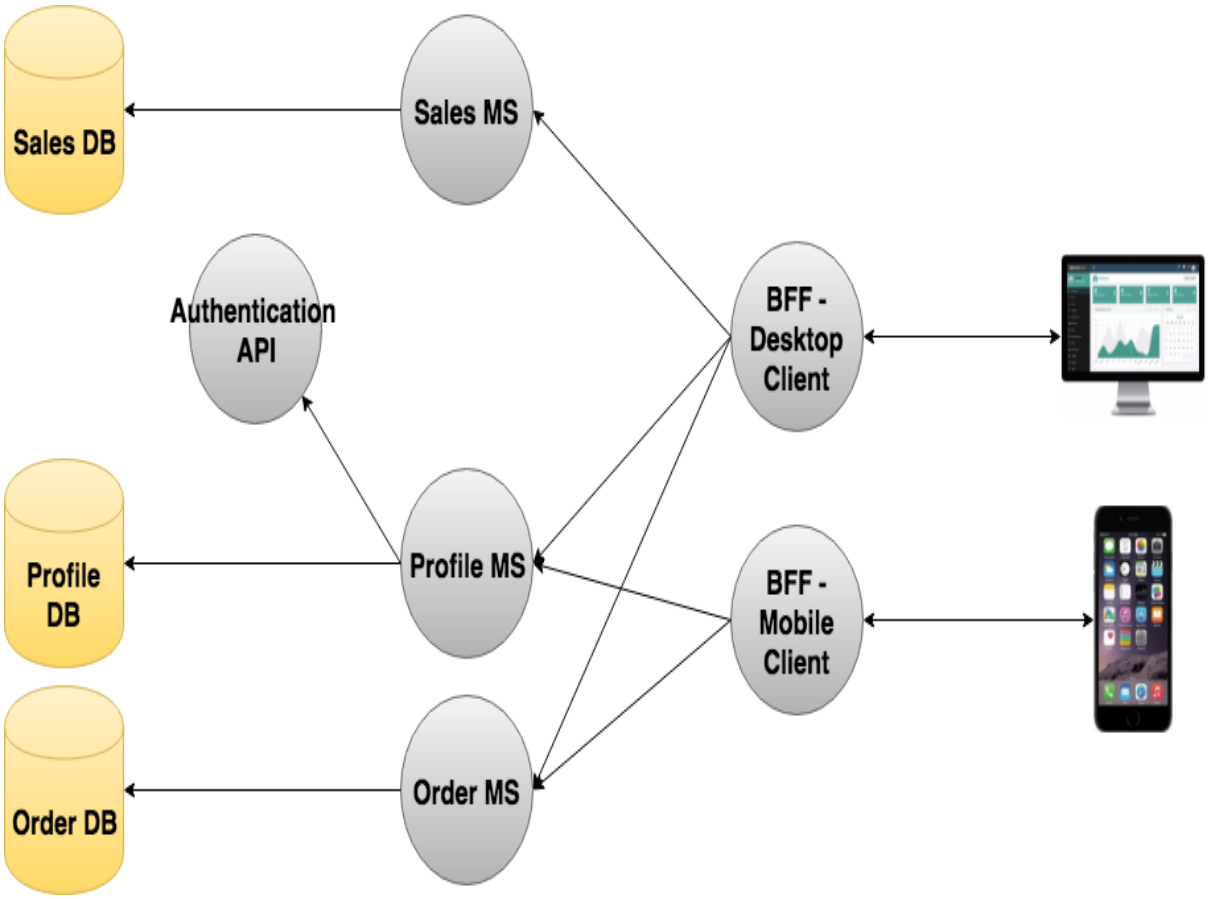

Figure 22: Inventory Management Application - High-level architecture design diagram with accepted user modifications

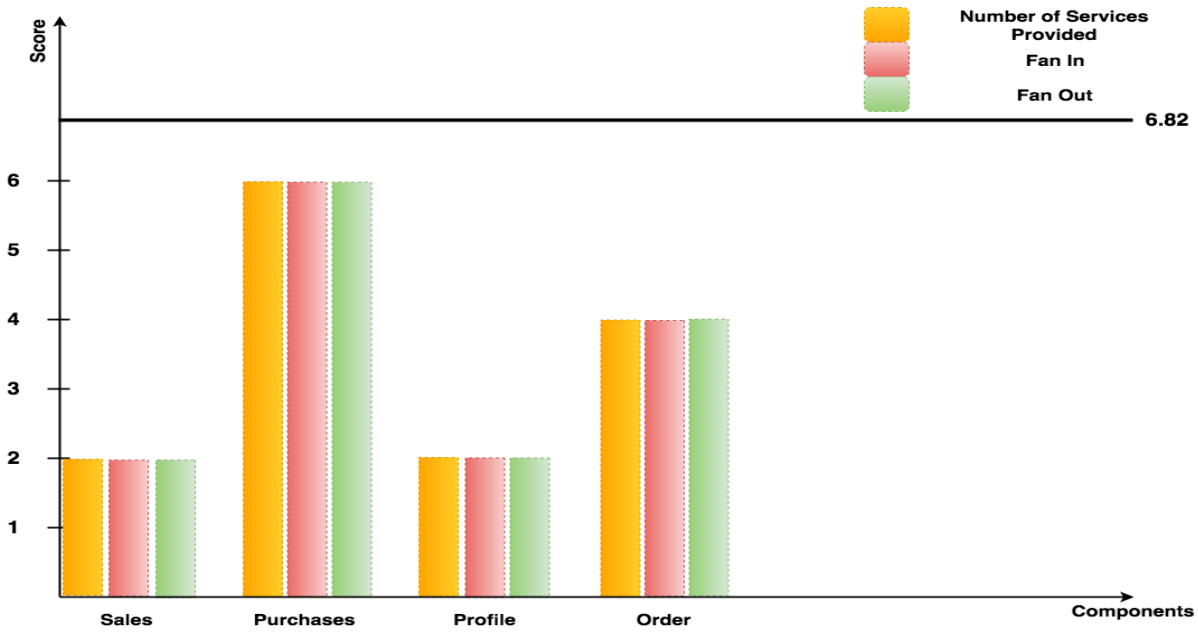

Figure 23: Inventory Management Application - Component based metrics evaluation for system generated architecture diagram

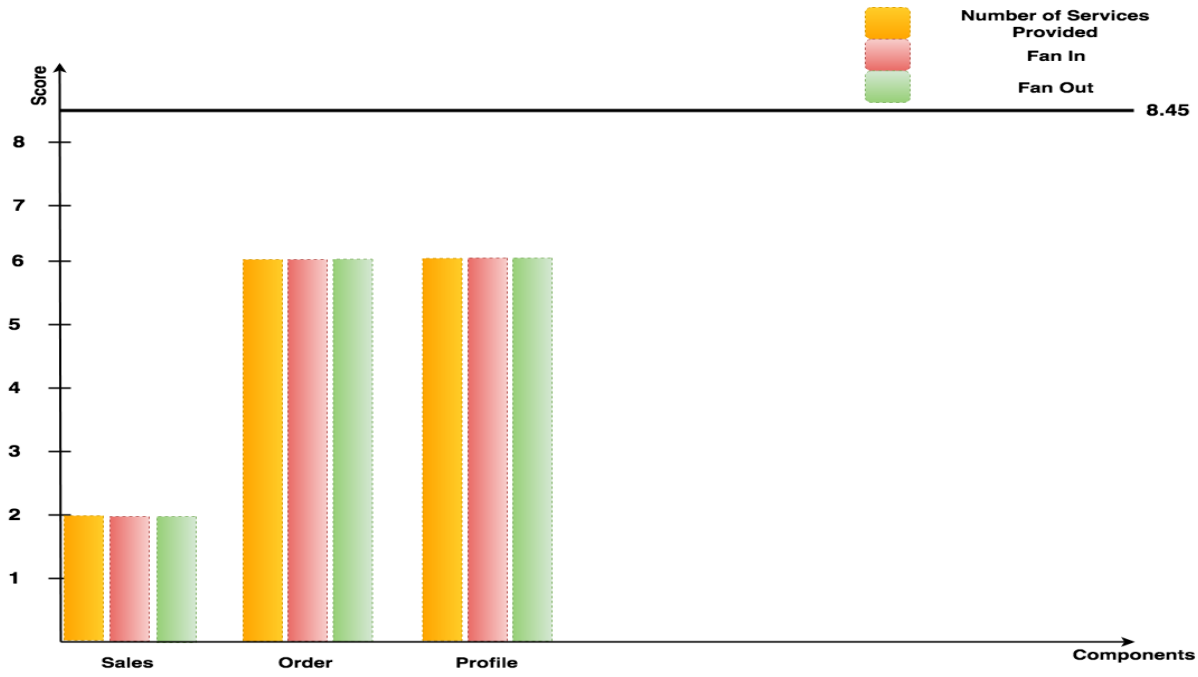

Figure 24: Inventory Management Application - Component based metrics evaluation on user modifications accepted architecture diagram

Table 8: Inventory Management Application - Services based metrics evaluation on system generated architecture

Metric	Value
Coupling	1.29 component calls per service
Depth of scenario	2 component interactions per service

Table 9: Inventory Management Application - Services based metrics evaluation on user modifications accepted architecture

Metric	Value
Coupling	1.28 component calls per service
Depth of scenario	2 component interactions per service

6.3 Performance Evaluation

As another experiment step we analyzed the performance of TheArchitect by evaluating the processing time it takes to generate the architecture diagrams for enterprise applications.

Table 10: Number of high-level system epics vs processing times

Number of high-level System Epics	Processing Time (seconds)
14 (Inventory Management Application)	1
33	2.5
57	4.5
93	7
113	8.5

Processing time listed in Table 10 include the complete timespan, starting from processing the system requirements up to the time tool finishes displaying the generated architecture. It is proven that tool has been able to speed up the traditional process of designing architecture diagrams, along with processing 100 high-level epics, in less than 10 seconds.

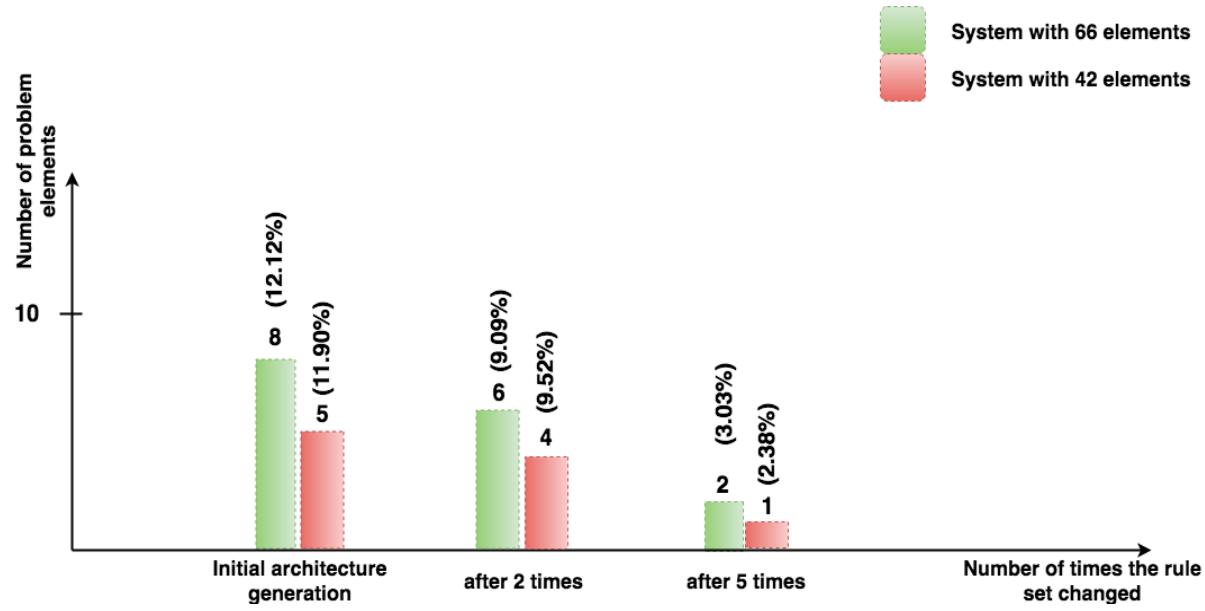

Figure 25: Number of problem elements against the number of modifications for the rule set

In terms of assessing the accuracy of the rule-based processing technique, conducted experiment obtained the statistics on how the number of problem elements got changed with the amount of fine tuning done on the domain specific rule sets.

Figure 25 shows how the number of problem elements reduces as we conduct expert fine tuning on the generated architecture. This was evaluated using two enterprise systems each containing 66 and 42 components. The statistics displays, that upon 5 modifications, number of problem elements could be reduced to less than 4% of the total number of elements.

6.4 User Study Statistics

TheArchitect was introduced to 4 different companies which operates within IT sector building enterprise software applications. We built up a pool of software architects and software developers consisting 50 from each category in order to conduct the user study.

Statistics presented in Figure 26 displays only 12 software architects and 6 software developer have voted against TheArchitect mentioning that they still prefer the traditional way of generating architecture designs. 76% of architects and 88% of developers who voted for

TheArchitect stated that it simplified and accelerated the process of generating a high-level architecture for a given system.

In summary, a considerable higher majority of both architects and develoeprs prefer to use TheArchitect in order to generate high-level architecture designs.

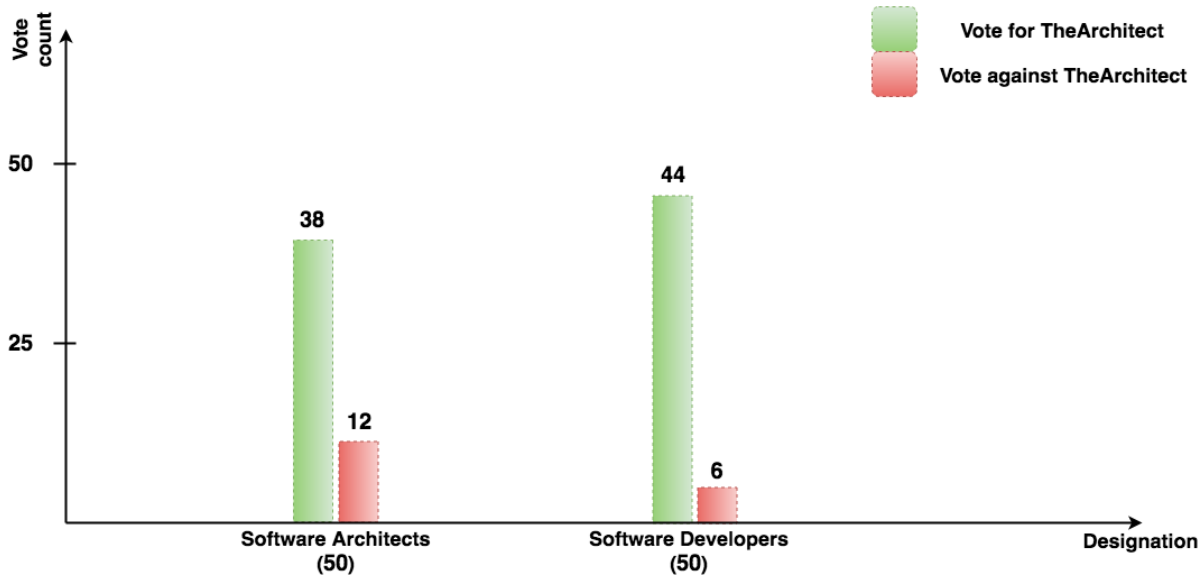

Figure 26: User preference statistics for TheArchitect

7. Conclusion

Software is critical in today's world. It is utmost necessary to get the architecture designed perfectly as all the subsequent development will be carried out based on the designed architecture. As explained throughout the document it is necessary to structure the architecture properly in order to deliver the application aligning with the customers expectation as well as adhering to the highest possible quality metrics. Current practice is that software architects design the initial architecture for a enterprise application considering the above facts, which requires high expertise and experience.

Software architecture is an area of interest that is subjected to content change. Currently the world is shifting towards designing and architecting serverless-microservices moving out from large monolithic systems. The traditional process of coming up with an architecture for a software system starts with the business analyst briefing the software architect with the necessary functionalities, outcomes of the intended system. The high-level architecture design that the software architect completes will be the foundation for the entire application. All the development decisions will be based on the high-level design hence it is utmost important to have a high-quality architecture design which takes into account the functional as well as non-functional requirements of the system. Generally, the above explained traditional process of coming up with an architecture design is tedious as well as could be error prone if not done with necessary expertise. As a solution this research proposes TheArchitect, a tool-based assistance for the architect in order to simplify and accelerate the traditional process of designing an architecture for a custom application.

We suggest TheArchitect, which is a rule-based system to auto generate serverless-microservices based high-level architecture designs for a given system, provided all the requirements are inserted to TheArchitect. As another important feature TheArchitect gives the opportunity to change the system generated architecture. As proved with experimental values TheArchitect maintains a superior quality in terms of the auto generated architecture diagrams. Further, the research also contains convincing statistics on the performance of TheArchitect in terms of accuracy of the processing technique and processing times.

Apart from creating a breakthrough platform for the software architects to escape from the traditional process of designing a high-level architecture into more efficient and accelerated mechanism it also allows any software developer to generate a high-level architecture for a software application minimizing the necessity of assistance of an experienced software architect. TheArchitect will process and map the system requirements provided by the business analyst or any other stakeholder who is familiar with the system requirements, into a set of predefined models. Next, the models will be passed to the architecture generator, which will process the models and infer the necessary architectural design for the intended application. Furthermore, moving back to the ongoing debate of serverless-microservices against large monolith applications, world seems to be moving along with serverless-microservices with the high value adding positives which it has produced. This is the main reason behind designing TheArchitect to generate best fitted high-level architecture designs which incorporates serverless-microservices.

7.1 Study Limitations

TheArchitect uses metrics-based architecture evaluation methodology in which it uses multiple metrics and weights for each metric. These metrics and weights need to be decided and determined by a panel of experienced architects. If incorrect metrics or weights come in to the equation it will directly impact the quality of the generated architecture. This is one critical limitation of TheArchitect as it requires human intervention and the entire evaluation depends on it.

If TheArchitect had not been used to generate an architecture designs for a specific domain, it would be using the base rule set hence it would need to go through few iterations to fine tune the rule set specific to that domain using experience architects' modifications.

Current version of TheArchitect does not consider non-functional requirements such as performance, security etc. in generating the high-level architecture. This would be another critical limitation when it comes to generating architecture designs for mission critical systems.

7.2 Future Directions

An important future directive is to improve the current architecture evaluation methodology to reduce/eliminate the dependence of defined metrics and its weights. Currently the designed architecture is only evaluated under a metric-based evaluation mechanism. This evaluation heavily dependent on the initial metrics that had been chosen and the weights used for each metric. Another future directive would be to conduct extensive research on the possibilities of incorporating another level of architecture evaluation schema. One possibility would be to evaluate more on scenario-based architecture evaluation methods such as SAAM, ATAM, CBAM, ALMA, FAAM etc.

Another future directive which could be taken upon in order to improve TheArchitect is to incorporate non-functional requirements in finalizing the high-level architecture for the system. Considering non-functional factors such as performance, security along with the functional requirements will allow the tool to perform better in terms of designing high-level architecture diagrams for mission critical systems.

Another future direction is to enhance the user experience in terms of modifying the system generated architecture. This will allow experienced architects to seamlessly engage with TheArchitect and effortlessly modify the system generated architecture.

8. References

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Proc. IEEE Intl. Conf. on Software Architecture (ICSA)*, April 2017.

[9] 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.

[10] 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.

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

[12] P. Clements, R. Kazman, and M. Klein, "Evaluating software architectures: Methods and case studies, Addison Wesley," 2002.

[13] 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.

[14] 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.

[15] S. Newman, “Building microservices: Designing fine-grained systems,” 2015.

[16] J.Gouigouxand 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.

[17] 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.

[18] A. Avram, “Faas, Paas, and the benefits of the serverless architecture,” 2016. [Online]. Available: https://www.infoq.com/news/2016/06/faas- serverless-architecture

[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] 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.

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

[22] 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.

[23] 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.

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

[25] V. Mezhuyev, “Architecture of software tools for domain-specific mathematical modelling,” in *Proc. IEEE Intl. Conf. on Computer, Communications, and Control Technology (I4CT)*, September 2014.

[26] C. Hu, F. Jiao, and C. Zhao, “An architectural quality assessment for domain-specific software,” in *Proc. IEEE Intl. Conf. on Computer Science and Software Engineering*, December 2008.

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

[28] M. Kajko-Mattsson, G. A. Lewis, and D. B. Smith, “A framework for roles for development, evolution and maintenance of SOA-Based systems,” in *Proc. IEEE Intl. Workshop. on Systems Development in SOA Environments (SDSOA’07: ICSE Workshops 2007)*, May 2007.

[29] M. Zuiga-Prieto, E. Insfran, and S. Abraho, “Architecture description language for incremental integration of cloud services architectures,” in *Proc. 10th IEEE Intl. Symposium. on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, October 2016.

[30] J. Wang, W. Chen, and H. Yang, “Architecture description language based on software reliability evaluation for distributed computing system,” in *Proc. IEEE Intl. Conf. on Computer Application and System Modeling (ICCASM 2010)*, October 2010.

[31] A. Smeda, M. Oussalah, and T. Khammaci, “Madl: Meta architecture description language,” in *Proc. 3rd ACIS Intl. Conf. on Software Engineering Research, Management and Applications (SERA’05)*, August 2005.

[32] Y. Zhenhua and C. Yuanli, “Novel architecture description language based on high-level petri nets,” in Proc. IEEE Intl. Conf. on Information and Communication Technologies: From Theory to Applications, April 2004.

[33] R. Kazman, “The essential components of software architecture design and analysis,” in *Proc. 12th Asia Pacific Conf. on Software Engineering (APSEC’05)*, December 2005.

[34] F. Garcia, F. Ruiz, and M. Piattini, “Metamodeling and measurement for the software process improvement,” in *Proc. ACIS/IEEE Intl. Conf. on Computer Systems and Applications*, July 2003.

[35] E. Hebisch, M. Book, and V. Gruhn, “Scenario-based architecting with architecture trace diagrams,” in *Proc. 5th IEEE/ACM Intl. Workshop. on the Twin Peaks of Requirements and Architecture*, May 2015.

[36] A. Patidar and U. Suman, “A survey on software architecture evaluation methods,” in *Proc. 2nd IEEE Intl. Conf. on Computing for Sustainable Global Development (INDIACom)*, March 2015.

[37] M. Babar and I. Gorton, “Comparison of scenario-based software architecture evaluation methods,” in *Proc. 11th Asia-Pacific Intl. Conf. on Software Engineering*, December 2004.

[38] K. J. P. G. Perera and I. Perera, “Thearchitect: A serverless-microservices based high-level architecture generation tool,” in *Proc. 17th IEEE/ACIS Intl. Conf. on Computer and Information Science (ICIS)*, June 2018.

[39] K. J. P. G. Perera and I. Perera, “A rule-based system for automated generation of serverless-microservices architecture,” in *Proc. 4th IEEE Intl. Symposium. on Systems Engineering (ISSE)*, October 2018