5-2019

Architectures v/s Microservices

Abhilash Nomula
anomula2901@gmail.com

Architectures v/s Microservices

by

Abhilash Nomula

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Information Assurance

May, 2019

Starred Paper Committee:
Dennis Guster, Chairperson
Lynn Collen
Sneh Kalia

Abstract

As it evolves, technology has always found a better way to build applications and improve their efficiency. New techniques have been learned by adapting old technologies and observing how markets shift towards new trends to satisfy their customers and shareholders. By taking Service Oriented Architecture (SOA) and evolving techniques in cloud computing, Web 2.0 emerged with a new pattern for designing an architecture evolved from the conventional monolithic approach known as microservice architecture (MSA). This new pattern develops an application by breaking the substantial use into a group of smaller applications, which run on their processes and communicate through an API. This style of application development is suitable for many infrastructures, especially within a cloud environment. These new patterns advanced to satisfy the concepts of domain-driven, continuous integration, and automated infrastructure more effectively. MSA has created a way to develop and deploy small scalable applications, which allows enterprise-level applications to dynamically adjust to their resources.

This paper discusses what that architecture is, what makes it necessary, what factors affect best-fit architecture choices, how microservices-based architecture has evolved, and what factors are driving service-based architectures, in addition to comparing SOA and microservice. By analyzing a few popular architectures, the factors which help in choosing the architecture design will be compared with the MSA to show the benefits and challenges that may arise as an enterprise shifts their developing architecture to microservices.

Table of Contents

Chapter Page

List of Figures

Chapter I: Introduction

Introduction

Service-based architectures (SOA) and microservices focus on services as their primary component to perform any task or functionality. They both share many similar features but differ in architectural style. The principal common object for service-based architectures is their distributed components, and remote access protocols control those components. Representational State Transfer (Rest) and Simple Object Access Protocol (SOAP) services are examples of remote access protocols helping the service components to communicate (Richardson & Smith, 2016). However, microservices have some specific benefits for businesses that follow the Agile methodology for developing and delivering enterprise-level applications. MSA was developed by adopting SOA, which focuses on breaking complex applications into small service components, active development, and testing practices to implement and remove centralized governance between elements (Richardson & Smith, 2016).

There are many characteristics of microservices which must be identified to gain a better understanding of architecting microservices. Through cloud computing, companies can deploy their application through a Platform as a Service (PaaS) or Infrastructure as Service (IaaS); consequently, they face many challenges to such as scaling of application, continuous delivery, increasing availability, high-level monitoring (Richardson & Smith, 2016). These reasons can force organizations to move their applications from a monolithic approach to MSA when they have multiple interfaces like HTML, Web services, and DB management. MSA offers many advantages over layer-based and monolithic style architectures. Components in MSA are self-

controlled and easy to maintain, which makes an application more robust, and help in developing modular applications (Gill, 2015).

Problem Statement

The primary purpose of this research is to identify factors that affect choosing an architecture by analyzing popular architectures and rating them based on these factors. At present, there is little conclusive research focused on the comparison of designs. The goal is to show how microservices have been developed to provide a solution to monolithic-based architectural problems.

Nature and Significance of the Problem

This research may help architects to gain a brief idea of different architectures, analyzing them with an array of deciding factors, and comparing them with MSA. Based on their application requirements, these architects can then choose the best-fitting architecture and provide the necessary steps to shift a developed application to MSA.

Project Objective

The primary objective of this project is to show how microservices are evolving, in addition to exploring and evaluating the different advantages and disadvantages of popular architectures. This research will provide the benefits of shifting applications to MSA from monolithic or other service-oriented architectures.

Project Limitations

Any limitations this project experienced was produced by real-time problems which can arise while designing an architecture.

Project Questions

What are the different types of architecture? Why is an architecture essential to developing an application? What are the factors to analyze an architecture? What are the reasons for shifting from a monolithic approach to microservices?

Chapter II: Background and Review of Literature

Introduction

New evolutions in cloud computing increase the chances of developing, deploying, and testing scalable applications effectively, which makes enterprise-level applications dynamically adjust their computing servers. These deployments create an unnecessary load on a server's resources, as they were not created to take complete advantage of those resources. As the monolithic application offers multiple services, some of which experience continuous demand. This can lead to inadvertently scaling an entire application, which may create unnecessary scaling on less-used functions. This phenomenon has proven to be the primary challenge in controlling the usage of server resources and effects the speed at which developments to an application can be deployed, which may directly or indirectly affect a business (Richardson, 2015)

As there is a rapid increase in several users accessing the internet, one application needs to scale for millions of users, which make companies develop their applications as Software as a Service (SaaS). The advancement in technology and high demand from customers' business are changing their models from B2B to B2C to provide better service so that a business may remain competitive. Government-based services are also changing to offer their services through the internet, which develops the application into a platform-based application to maintain their facilities in the case of heavy demand (Buyya, 2010).

What is Architecture

Put simply, architecture is a significant collaboration or grouping of a system embodied within the components and defining the relationships between them concerning the environment.

In other words, architecture is a guide of principles in designing and developing the application, or a complete collection of components organized and assigned a specific set of tasks to perform. Here, the system defines individual applications, regular systems, sub-systems, product lines, or whole complete enterprise. Systems perform multi-threading to accomplish this. This environment defines the setting and circumstances in every phase like development, testing, operational, political, and any other type of influences (Hammouda, 2004).

A mission could be an organization idea or a service or an intention of a stakeholder to fulfill some set of objectives. A component can be logical, technical, technology-independent, small or large-grained. The content wrapped within the component can be modified or replaced within its environment. Architecture is also considered a reusable solution to recurring problems so that an architect can efficiently understand how each major component affects the system and how messages should be communicating between them. It's all about making standard decisions of how to organize the system, like the selection of elements and their dependent interfaces by which a system can be composed and bind together to make them behave as per the business requirements.

The design of a program or a system where elements are comprised exhibits the logical or physical property implementation in the architecture. It gives a clear picture of how to assemble the components, which can be a class, interface, or subsystem. The designing phase of an application impacts the design decisions of how elements in the system behave and communicate. A better architecture can make a better and successful application. It also makes developing, implementing, and testing phases easier. It provides a blueprint for developers before giving functionalities to the components of the application, or it can create a more detailed

outline for fulfilling assigned commitments effectively. By going through their designs, developers can easily find issues within their system (Hammouda, 2004).

Key Roles of an Architecture

- **Defines structure.** System patterns define characteristics of architecture, like diagrams specifying structural aspects such as layers, elements, or components. Structural aspects are understood in many ways and most of them are vague as a result. Processes, databases, supported libraries, and element nodes are all considered structural elements. Along with defining individual structures, it also defines the relationships, interfaces, and partitions of these structures. It defines which code class needs to be a parent class and which needs to be dependent (Shaw, 1994).
- **Defines behavior.** Along with the structure, it defines the interactions between them which decides the behavior of the system. Behavior defines a property or functionality of the class, which can perform alone or possibly provide some data to other dependent components.
- **Focuses on significant elements.** Though architecture defines structure and behavior, it does not need to be concentrated upon only these two subjects. It mainly focuses on elements which are significant. These elements contribute a major part to an application which has a long-lasting relation. They are essential and possess vital behavior along with significant qualities such as reliability, scalability, and usability. Architectural significance can be considered as cost-effective, as its major concern is to make decisions in choosing one element over another or to make an update on them which includes cost. As it focuses on major elements, it gives a clear

perspective of the application or the system, which is most relevant to the architect. It also helps the architect to manage time and space complexities.

These significant elements are not static and can be affected by changes over time, changes in requirements, identified risks, performance concern, or even difference in opinions. An architecture can be deemed effective when it maintains stability in terms of execution at the phase of change. If it is not stable and experiences a continuous change to fit the minor changes, then it will be considered as an inefficient architecture (Shaw, 1994).

- **Balances stakeholder needs.** The main agenda of building architecture is to address the needs of the stakeholders who have invested in, or are dependent on, the application. Sometimes it is not possible to meet every requirement specified by the stakeholders, mainly when it involves functionality and timeframe as these two are mutually exclusive. Architecture should be either reducing the scope of its requirements to deliver on time or increasing the time to cover the complete scope. Multiple stakeholders have multiple opinions, and those opinions can be opposed. Maintaining an appropriate balance will resolve the conflict in this situation. Making trade-offs, negotiations, and balancing are the essential characteristics of an architect who can impact an architecture. Below are some of the needs of a stakeholder, who is considered as an element impacting the shape of an application.
    - Considering end-user intuitive and analyzing expected behavior, performance, security, usability, reliability, and also availability.
    - End-user behavior.

- Analyzing tools, monitoring systems, administrator micromanagement behavior, and the process of administration.
- Marketing which considers cost, competitive features, products.
- A simple and stabilized approach is expected from the development side.
- Project managers worry about the estimation of time, scheduling, release dates, and also being productive in terms of resources and budget.

Non-functional requirements can also affect the design and challenges of an architect's abilities. While they may not represent the system qualities or constraints, these requirements have a crucial impact on the designs (Shaw, 1994).

- **Based on rationale.** The main aspect of architecture is itself. Therefore, it is always important to document an architecture's design, the decisions which impacted the architectural style, and the motivation behind those decisions. This documentation is valuable not only to stakeholders but also to an architect to help avoid unnecessary retracing of steps when questioning a decision's rationale. This behavior helps an architect defend and justify their logic when their architecture is under review.
- **Conforms to an architectural style.** Most architectures are derived from systems with similar sets of concerns, and these similarities can define architectural patterns. Developing a complex pattern or a compound pattern (many patterns aligned in a process) is a good practice for an architect. A single system can exhibit multiple architectural styles, examples of which include distributed styles, pipe-filter style, data-centered style, and rule-based style. Architectural styles generally define how components, elements, connectors, or constraints will combine into a system.

Architectural patterns make an architect's work easier, as these patterns are documented in terms of the rationale behind its use. Documentation helps in understanding an architecture's properties (Eeles, 2004).

- **Environmental influence.** All systems or applications need an environment to run in, and a chosen environment may have a significant impact on the architecture, sometimes referred to as "architecture in context" (Gill, 2015). An environment decides the boundaries where an application should run, which impacts the architecture. The factors that influence architecture are business requirements and standards, stakeholders, some external technical aspects (application integration to other systems), and internal technical constraints like organizational standard.

  Vice versa, architecture itself influences the environment either by improving the technology or by creating reusable assets to the organization involved in the development. It also impacts the skill set available or provided in the organization. To use the software, it needs some hardware for executions, which is why the environment considers a primary component for the architecture (Gill, 2015). The developed and executed application is a combination of software and hardware, and this combination makes the application more usable, reliable, and securable; software alone cannot achieve these properties. In other words, we can say the system which is running and performing the desired tasks is a combination of software, hardware, multiple processes, and people.

  In terms of increasing the performance which is the important functionality of the application, there must be an increase in the system elements attached to hardware

rather than software. In some other situations, to decide on improving the system usability, there would be a requirement of the interface which is a human being rather than specific hardware or software. Subsequently, the best achievable results require a combination of people, hardware, and software. It is better to treat every element equally rather than thinking software is lesser compared to hardware and assuming that it is just a requirement to make hardware work perfectly, or thinking hardware is a second-class element which is required to execute the software (Shaw, 1994).

- **Influencing team structure.** An architecture can define a coherent grouping of system elements which addresses concerns. For example, a payment processing system's architecture should define each grouping of elements such as account information, customer information, order processing, communicating with external and internal elements, integrity, and security. Each element requires a different skillset so that a team can be built and structured once the architecture defines every element in the system.

  In some cases, architecture is also influenced by the initial team structure, but this should be avoided because architecture is more important and ideal than satisfying the team structure. The architecture will also reflect the organization means and behavior. But this showcasing view is not a good practice and should be avoided for practical reasons. Skillsets and availability of resources can represent what the constraint is, and this must be taken into account while building an architecture (Shaw, 1994).

- **Presence in every system.** If a system is simple and consists of a single element, it is not worth placing an architecture in that system. Informally documented architecture

can also be considered in the same way. The process of documentation and recording the decisions impacting an architecture tends to be well-thought out, as an undocumented architecture is far less effective. To prove that the system meets the standards and requirements of usability, maintainability, security, and reliability, an architecture needs to be properly documented. Missing architecture documentation is considered a major concern but tends to be accidental (Eeles, 2004).

- **Architecture with a scope.** There are many forms of architecture other than software architecture, like application architecture, enterprise architecture, information or communication architecture, infrastructure architecture, and hardware architecture. Every architecture has its specific scope and activities, as there are no industry standard definitions for forms of architecture. One form may have multiple meanings and multiple forms defined in the same way. Rather than being defined with an industrial standard, the relationship between these forms has instead been defined with the scope of the terms inferred in the below figure.

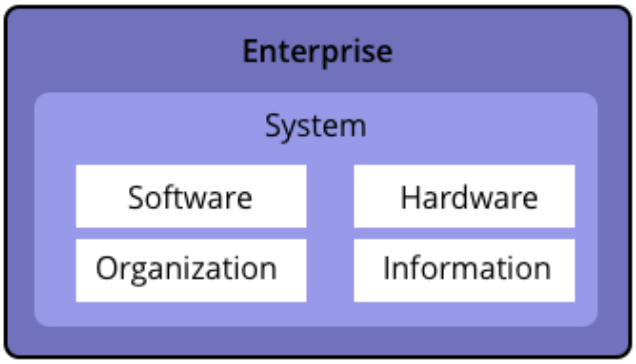

Figure 1: Scope of architecture (Hammouda, 2004).

These four architectures shown in the above figure can be treated as a subset of system architecture. From this figure, we can observe the following points:

- Software architecture should have documents related to how the software should perform its tasks.
- Hardware architecture considers components such as memory allocation, CPU, processors, servers, printers, and some input hardware materials and how they are related to each other.
- The organizational architecture consists of business logic, process, management, and roles and responsibilities of individuals in the team.
- Information organization consists of information-related structures and how to communicate with these structures, as well as security measures to be taken while passing confidential information through an organization.
- Enterprise architecture is a superset of all the architectures, along with the individuals who have a direct link with the business activities and objectives. It also covers business agility and can define the efficiency of an organization. It may cross the boundaries of the organization.

There could be many other definitions for these different form of architectures, but major organizations accepted more or less the above terms (Eeles, 2004).

Problem with Monolithic

In this type of architecture, the whole application code is developed and deployed in a single project. This is due to the fact that enterprise-level applications need multiple developers, and whenever they want to update a single server resource they need to make sure that other services are not broken and will run smoothly. These transitions require a lot of time and human resources to maintain constant functionality. Due to competition, businesses want to add new

features to their applications, which increases the nature of complexity in this type of approach and limits applying their innovative skills into the application. When a new update is attached to a service to make an application run into production, whole sets of servers need to be restarted. This impacts the user experience, which is not at all a good practice in this modern situation.

Monolithic Deployment

Monolithic applications are single code-based services, meaning they shut down complete services if one application service fails, also known as single-point collapse. Many limitations of monolithic deployment have led technical experts to discover solutions for cloud computing problems. As many large online-based companies face these failures, they have started creating strategies and patterns to reduce the complexity of applications, which is known as a lightweight subsidiary or system-oriented architecture. With the change in the structure of deploying the cloud-based applications, these companies were able to reduce build failures, scaling computing services, and now have more control over time management and team management. Many reputed organizations like Amazon, LinkedIn, and Netflix are following this new architecture pattern to alleviate problems caused by monolithic deployment, to gain better user experience, and for scaling of enterprise-level applications or products (Lorido-Botran, Minguel-Alonso, & Lozano, 2014).

This new pattern is known as a microservice architecture, which has the influences of SOA. There have been many discussions on how to consider this new architecture, as it consists of a new innovative software pattern or a pattern inherited from SOA. Though it looks like a child of SOA, years of industry-level changes to this new style of deployment have reduced the complexity, focused on agility, and provided more services to increase developer productivity. It

also reduces problems from using centralized ESBs, providing effective computing solutions in cloud technology to millions of users (Fowler, 2014).

Technical Issues

This section will compare architectures using a real-time example application for a new taxi-service similar to Uber or Lyft.

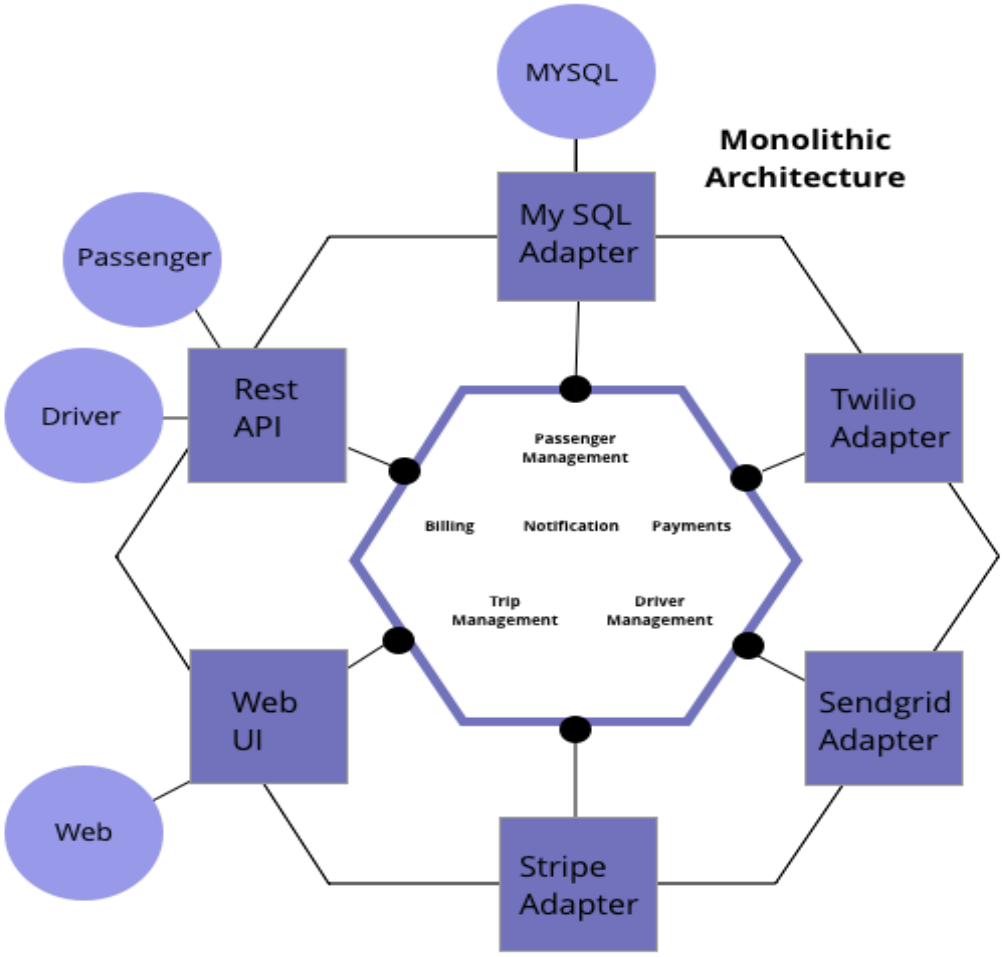

Figure 2: Monolithic architecture (Richardson, 2015).

This application has multiple functionalities like customer management, driver management, charges information, payment management, routes/map integration, and notifications. We need web services, UI services, database services, and adapter services like

Twilio, SendGrid, or Stripe to manage the application (Richardson, 2016). In a monolithic approach, all the functionalities have bound these services into one whole set of single code-based application. This project can be created using any language and framework and deploy to any server. If the application is built in Java, it can be wrapped in WAR files or self-contained Jars and deploy to the server. Directory hierarchy can be used for Ruby on Rails and Node.js. As the IDE's supports single code applications, they are easy to develop, deploy and test. Complete UI testing functionalities can also be performed with the help of a testing package called Selenium. These types of practices simplify the work of deploying and scaling by running the application copies in a load balancer during the initial stages of the project (Richardson, 2015).

However, the problem starts when the product becomes successful, and this simple approach creates a variety of limitations. Successful applications require more changes in terms of performance and additional features that eventually make the application big. There will be new stories lining up in every sprint, which means pushing new code into the existing application. After some sprints, the application will increase in size and therefore increase the complexity of the application with the growth of dependencies. This creates a lot of pain for Agile developments, and small mistakes can lead to a big loss. Another issue with this style is an application is that it consumes the time and efforts of developers to completely understand the code in order to update or add new services to the application, which represents the negative attitude impacting fixing bugs and expecting effective functionality of the change. If high efforts are required to debug, the application may become unintelligible (Continuous Delivery, 2012).

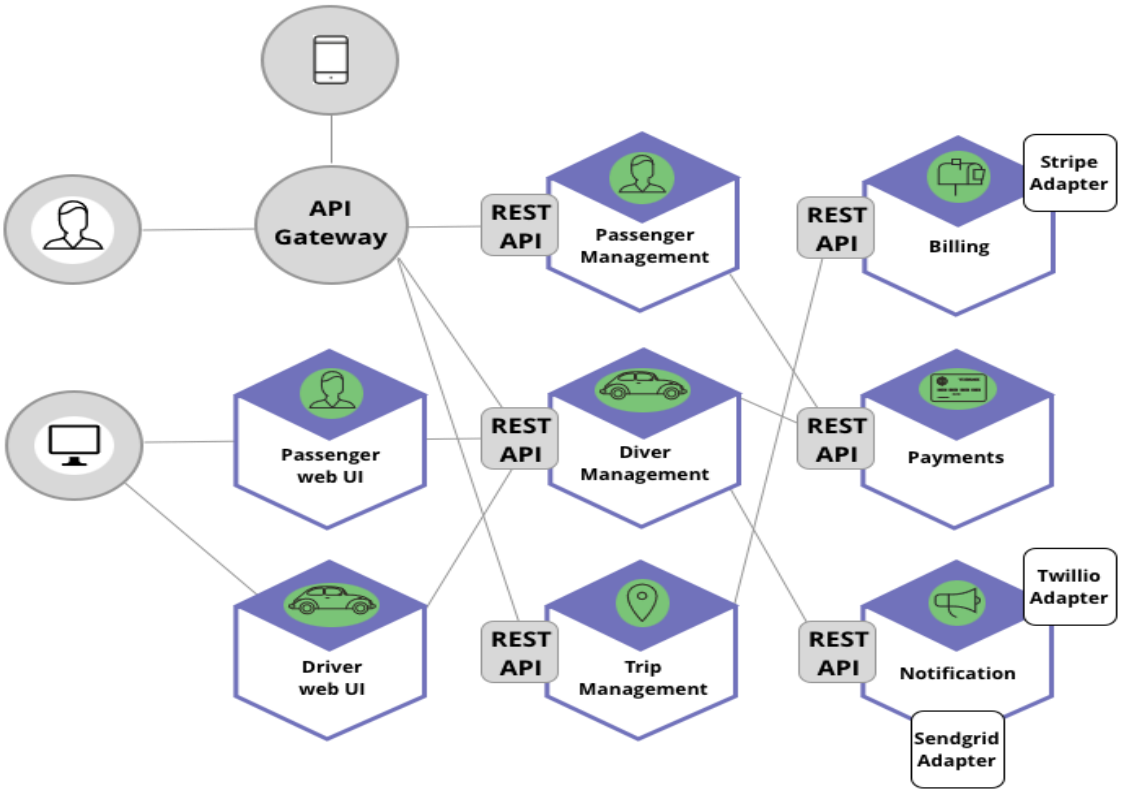

Figure 3: Microservices architecture (Richardson, 2015).

Reasons for Shifting

Deploying all the services into a single module will not only slow down the development but will also affect performance while starting the application. Sometimes it takes consumes considerable amounts of time, affecting human resource productivity if the application needs regular start-ups. Complex applications developed in a monolithic pattern creates issues when code changes need to be pushed to the server regularly, which in turn needs complete deployment if one service must change or update. Sometimes continuous redeployment affects the running of the application and needs to put more effort into debugging or testing the product when this issue cannot be easily identified.

This architecture can also raise conflicts when choosing hardware. As multiple services are deployed into a server, some modules may need compute-optimized instances while others may require memory-optimized instances. Whichever kind of instance a module might need is dependent on the type of processes, like image processing or database storage (Gill, 2015).

Another main concern regarding the monolithic approach is reliability. A single vulnerability in an application can affect the application process, which can slow down the system or even shut it down. Since one service impacts the other services, and each instance is similar, it challenges the availability of the application. When a business wants to shift their applications to another framework or platform, utilizing a monolithic methodology increases the difficulty level for modification and can force a business to invest more time and money into making the new framework or language simple and effective. Which in turn, helps to create a bridge when a business wants to adopt new changes in terms of technologies or the structure. If businesses want to invite a new set of developers into a project, time and effort are needed to train them. However, the issues of scalability and reliability remain a pressing concern (James McGovern, 2006).

Chapter III: Methodology

Introduction

Building an architecture to utilize services is called service-based architecture. Service-oriented architecture (SOA) and microservices fall under this type of architecture, where services are considered as the primary architecture component on which functionalities are developed. There are many common characteristics between both SOA and microservices; one among them being distributed pattern, which allows components that support services to be accessible through protocols allowing remote access. REST, SOAP, and AMQP are some examples of remote access protocols. These service-based architectures have many advantages in terms of scalability and decoupling, in addition to giving more control during development and product testing. The distribution pattern in these architectures makes them controlled and easy to maintain. As these were loosely-coupled, they make the applications robust and responsive (Richards, 2016).

Architectural Patterns v/s Design Patterns

Architectural patterns are similar to design patterns, but the scope of each pattern is different. A design pattern will mainly concentrate on the style of the code and focus on a small range of an application, whereas an architectural pattern will focus on the large scale and decide how elements should behave and perform their tasks. Simply put, architectural patterns define a global structure, while design patterns define the style of an internal structure. A design pattern is considered a problem-solving technique for recurring issues faced by developers or programmers, and are basically developed from the ideas and experiences of programmers. As these patterns maintain their predetermined standard, the code will appear cleaner, simpler,

organized, and easier to understand. It also makes information easy to digest and subsequently helps to train non-technical personnel.

Both design and architectural patterns have similar views. A design pattern explains the implementational view of an idea, while an architectural pattern explains the abstract view of an idea. A design pattern will picture the detailed implementation of specific elements or domains, while an architectural pattern will picture the abstract view of an entire application's functionality. Architecture designing is the first step in building a system where decisions regarding how the components are divided into questions of how behavior can be injected and how the components can be connected to perform a specified task. A design pattern is introduced once the architecture has been finalized; in this phase, all the templates designs, code structure, and components are built. Architecture decides which design pattern suits the system best (Franchitti, 1992). Below are examples of popular architectural patterns:

- Model View Controller
- Model View Controller Service
- Multi-Tier
- Event-Driven
- Service-Oriented

There are 26 design patterns, which can be categorized as shown below:

Creational patterns. Includes Abstract Factory, Builder Factory, Object Pool, Prototype, and Singleton Pattern Factory (Franchitti, 1992).

Structural patterns. Contains Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Private class Data, and Proxy pattern (Franchitti, 1992).

Behavioral patterns. Includes a chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template Method and visitor pattern (Franchitti, 1992).

Based on the problem, choosing a pattern which best fits an application requires a considerable amount of experience. The pattern-style of coding can be implemented in any field of computer science from basic application to high-end AI-based applications, from games to business intelligence, and from websites to embedded systems. While selecting a pattern, one needs to be careful as selecting an ill-suited pattern can cause difficulties in reading and understanding the code. The problem must be considered first to decide what steps are needed to solve it, which helps in choosing a suitable design pattern. For implementation, there are no specific rules which allow customization to obtain better results. Control over the patterns must be maintained to follow the program (Franchitti, 1992).

Styles or patterns. Patterns can be described as a way to define a solution. Styles are the names given to designs. A single pattern can have multiple styles embedded in it, and one style can make use of multiple architectures. A pattern can be considered a subset of a style to solve a specific problem or scope. Architectural style mainly describes the features of the elements involved and their forms, which can make a stabilized application. Low-level patterns which specify the programming language are called Idioms and can help implement aspects of the elements and relations between them (Franchitti, 1992).

Why Architecture is Important

Architecture is very important for any application to maintain scalability. It is even more important if a team consists of more programmers, as there will be continuous updates,

developing new features, and multiple approaches. Giving more importance to architecture can make the code organized, readable, and easily updated while also stabilizing the environment, improving performance, and assuring an application is of good quality. All the programmers or developers in a team should have a clear picture of the architecture to help them understand how to develop their application, ensure they have the same view of the application, and avoid any conflicts. Without architecture, the application can face many issues, and code can be incorrect and require more effort to update a simple functionality. An application without architecture can be considered an undisciplined system while making it hard to analyze issues and identify problems in the application. Architecture is nothing but a set of development rules all members of a development team should follow, and every feature should be written and designed in a consistent way that describes the architecture. Implementing an architecture not only helps build a strong and efficient application but also helps one team member to understand another's intentions and problem-solving techniques easily. These rules should be described and decided by team members with more experience and who have the ability to guide their teammates to maintain consistency across the application.

There will be many aspects influencing the decision of choosing and implementing the architecture. Choosing the right pattern to act as the interface between databases, business logic, and the user interface is the key part of the architecture. This pattern will decide how the code will look and gives an idea of what components are attached and also what elements to be considered in development and how exactly they should behave and connected. The main advantages of having an architecture are described below (McGovern, 2004):

Stakeholders communication. An application following an architecture-based structure will have a common abstraction. This gives stakeholders a common view, mutual understanding, effective communication, and helps them make quick and effective decisions. Every stakeholder for a project exhibits different characteristics. The system user, end-user, project manager, programmer, and tester can be affected by their architecture. For example, an end-user can be worried about the reliability, usability, and availability of the system. The product owner can be concerned about their budget and deadlines. The manager can be concerned about the architecture allowing the team to work independently and in a well-disciplined way. The tester can be worried about the testing scope allowed by the architecture. The architect can be worried about implementing an effective architecture to obtain overall control and may strategize how to achieve their desired goals.

For all the concerns expressed by the stakeholders, the architecture provides a common language which can communicate with each stakeholder, which helps them identify problems and find solutions to even large and complex systems. Without such a wide-spanning language, it becomes difficult to understand the functionality of the complex system. Effective communication increases the quality, usability, and reliability of an application (McGovern, 2004).

Decisions on designs. In the software development cycle, designing comes first. Early decisions on designs have greater weight when compared to the system development, while late decisions may come after development starts and could create issues that impede finishing the project before its deadline.

Architecture represents a set of early design decisions. These decisions are hard to update or change in the process of development and can create a large impact throughout the cycle. Decisions described by the architecture reflects the structural design during the implementation of the system. Each component of tasks, which are implemented and broken into components during this process, should perform their respective work and need to communicate with each other.

Implementation considers resource allocations which might be invisible to the implementors who develop the system. The separation of concerns allowed by the design makes management decisions look like the best judgments were made to improve the personal and computational process. In this process, individual builders need to concentrate more on constructing specific individual elements rather than overall architectural structures. The responsibility of architectural trade-offs need to handled by architects, and they need not have complete knowledge over the algorithm of the application (McGovern, 2004).

Transferable abstraction of the system. Architecture in software development constitutes an effective and intellectually graspable model while giving structure to the system and directing elements on how to work together to perform the desired task. This model of one system can be transferable and applied to another system which shows similar qualities of attributes and functional requirements, as well as promotes largescale re-usability. Architecture determines the ability of elements used to build a system to exhibit the system's quality attributes. Some of the quality attributes that should be considered before building architecture are listed below (Lindgren, 2008):

- If there is a large scope for modifications, elements must be built in such a way that changes to their responsibilities will not have far-reaching consequences.
- Inter-component usage needs to be managed carefully if the project delivers continuous sub-systems.
- Restrictions over elements which are coupled internally can make them more independent; when they are extracted to be used on other systems, they do not acquire the attached files which were coupled to them. This reduces the performance load on the application.
- If scalability has more priority, usage of local resources should be carefully handled in a way that higher capacity elements can be introduced into the system.
- Managing time-based behavior of the elements. Frequency of communications between elements should be considered if a high-performance system is expected.
- If a system should be highly secured, communication between its elements needs to be protected, which may unintentionally give access to confidential information. Building security walls and introducing trusted kernels can also increase the security of the system.

The above strategies can be implemented at the architectural level. However, introducing these strategies alone cannot give complete assurance for building a fully functional and high-quality product. Improper architectural design and incorrect implementations of the required strategies can also lead to failures. All decisions from design, implementation, development, and effective programming are major concerns of quality. Therefore, quality is not handled by the

architecture alone, but proficient architecture is always required to gain a high-quality product (Lindgren, 2008).

Dictates organizational structure. Mainly architecture is involved in prescribing the structure of a system. This structure matures into a development project, which in turns becomes the shape of the organization. The larger systems are divided into small groups of different proportions and assigned individual tasks known as work break down. In the same way, architecture decomposes the large system into small units where planning, budgeting, and scheduling are decided with concerns of internal communication channels, controls configuration, integration, migration of technologies, and test plans kept in mind.

Integrated applications communicate through channels or interfaces specified by major elements. One main disadvantage of work break down is that some dependent elements will be blocked or frozen by the architecture. If one subsystem holds responsibility and is formalized into a contractual relationship, updating them can include more cost. Even tracking the statuses of these elements can also be difficult. Architecture is an agreement between the stakeholders, which becomes almost impossible to change or update because of alterations in business and management requirements. Therefore, proper discussions and evaluations are mandatory before signing off on the agreement on architecture. The proper evaluations made on architecture can give more accurate predictions on the quality of the application (Lindgren, 2008).

Provides effective reasons for change. As per the recent studies, it is a well-known fact that major investments on an application occur after the initial deployment of the system. The number of people involved in the development will increase after the initial deployment, as there will be more requirements added to gain a better idea around the system’s functionality. Many

programmers or developers generally work to reconstruct existing code rather than the system during this new phase. Software systems are often updated, and satisfying the requirements will oftentimes become difficult.

An architect divides the new updates or functionalities into three categories: architectural level, non-local changes (integration), and local (internal changes). The local updates require a change in a single element, whereas non-local requires an update in multiple elements while maintaining the underlying interaction between elements which are described by the architecture. However, updating the architecture involves most of the elements and can, therefore, impact the contact between the elements. In other words, the major system functionality will be affected. Consequently, when a change is decided to be essential for the system, the least risky path for updating the components must be selected while considering the impacts of the proposed changes. When determining the consequences and priorities that have overall sight into the relationships, performance, and behavior of the elements, insight is necessary before decisions regarding the proposed changes can be provided by reviewing architecture (L. Bass, 2003).

Provides evolutionary prototyping. Defining architecture and documentation helps in building the prototype of the system in the following ways:

- In the early phases, the system will have executable elements, which then mature into prototype parts and are replaced by fully developed versions of software components. These prototypes are considered low-maintained versions of software which perform tasks in the desired way.

- Having executable prototypes during the early stages can help identify any potential issues or problems, which can then be easily resolved before progressing to later stages (Bass, L., Clement, P., & Kazman, R., 2003)

Building architecture can reduce the risk of errors and guide the system down the right path. Considered as part of the systems, the development costs involved in building the framework are distributed to overall systems (Bass et al., 2003)

Gives accurate cost and scheduled estimates. Managing costs and estimating schedules are important roles of a manager to meet their deadlines and gain the resources required to build their assigned project. Estimating costs for individual elements will be more accurate estimations for the overall system development. Aligning these elements properly to find the total costs is provided by the architecture. Elements are distributed across teams and can assign ownership to them, which results in accurate estimations than the overall estimation from the manager.

Architecture increases the probability of reviewed and validated system requirements. Applying re-usability features into the architecture can save a considerable amount of money and time. Re-usability in architectures is one of the advantages with the systems having similar system components. Re-using functionalities, structure, and code can make the application simpler and easier to understand. When multiple similar systems are involved in an application and re-usability needs to be implemented in the architecture level, it should be decided and carried out in the early stages of the designing architecture (Bass et al., 2003).

Allows integration with large external systems. Architecture-based development mainly focuses on developing individual components and assembling them, which are independent of each other. Development is measured based on the composition, whereas,

without architecture involved in the development, paradigms are measured based on the line of code. Development of individual elements is possible because the architecture describes each element in the system. These descriptions explain how each element communicates, shares controls, and what type of data can be consumed and returned along with the protocols for sharing the available resources. The key is organizing element structure, interfaces, and operating components (Ruhe., 2005).

Contracts in service-based architecture. Contracts in service patterns define how communication or data transfer between remote service and the client request happen. They typically communicate in XML, JS, JSON, languages and treat each element or component as an object. These contracts cannot be labeled as a lower priority, as they have a huge impact on building the application. These contracts can be known as difficult tasks, as they need high accuracy, a special team, or high observation in these types of architectures. Different models can be used in this architecture, such as customer-driven models or service-based models, and the only difference is the degree of collaboration. These collaborations give contracts the freedom of evolution with or without considering the need for services expected by consumers. Service contracts are considered as a single owner who makes the consumers adopt its functionality (Ruhe., 2005).

Data Analysis

- **What Determines a Good Architecture:**

As people are different-minded, architects from the same organization can create different architectures. So how can an architect determine which one suits their application? There is no such thing as a good or bad architecture; each fits the requirements of a different

application. For example, a layered architecture may be suitable for financial management systems but not for an avionics type of application. High scalability or high modifiability alone cannot decide the prototype of the design. Below are some general rules to be considered while building an architecture. These rules are classified as process-related rules and structural rules.

Consider the OASIS reference model. Components are described as a functionality where more than one service or capabilities access in SOA. This access is achieved through an interface by following exercised policies consistently, which define the service description. With defined interfaces and contracts, customers of the business can access expected service. To define a service contract, one should have a better understanding of the characteristics of the architecture. This characteristic is different in microservices, though they rely on service as a component. SOA and microservices can be classified and compared on how they coordinate with the service owner, pattern classification, and service granularity. The services are classified based on the taxonomy with service type and business type. Service type gives the role of service in how they implement functionalities required by the business, and they sometimes help build non-business functions such as audits.

Securing and logs are defined in the architecture pattern level. The business type defines the area where it plays its functionalities related to business, such as order processes, reports, graphs, and trading procedures. This is well-defined at the implementation level. Patterns followed by architectures are basic points for understanding service types, which are easy to modify based on their requirements (Papazoglou, Troverso, Dustdar, & Leymann, 2007).

Process rules of thumb. The architecture should have either one owner or be owned by a group or a small team led by an identified leader.

- The team needs to identify its prioritized quality attributes and should have all the functional requirements which can satisfy the application.
- It should be documented in the static and dynamic view and can easily be understood by non-technical stakeholders.
- All stakeholders should actively review the architecture.
- The architecture should be analyzed based on quantitative measures and identify quality measures in the early stages.
- The architecture should start in a small skeleton form and the communications between the components are exercised. Adding functionalities later can make it more stable. This process can reduce integration and testing efforts.
- It should return as a set of solutions to resource contentions. These solutions need to be identified, specified, maintained and circulated. If performance is the concern, the architecture needs to guide the development teams on how to balance the load to get timely responses from the server (Kazman, Klein, & Clements, 2000).

Structural rules of thumb.

- The architecture should be described as a defined set of modules, where each module is allocated with functional requirements based on the principles of hiding and separation. Modules with hiding responsibility should also be included in the encapsulation of computing infrastructure.
- A well-defined interface that encapsulates and hides its changeable aspects from other modules which use its functionality can create a strong hiding module. These types of modules allow the development team to work independently.

- By using well-known architecture tactics, one needs to identify the quality attributes.
- Architecture is implemented in such a way that it should have less dependency on the commercial tool or product. In the case of changing the product, it should be straightforward and reduce expenses.
- Data production and data consuming modules should be independent, as these are the modules most affected by change. With alterations to the requirements, less dependency can help the development team upgrade the modules stage by stage.
- Parallel processing systems should have well-defined processes and should not resemble a mirroring decomposition structure. A single process initiated through multiple modules may experience various methods that invoke a unique operation in each module.
- Every task must be documented so that assignments are editable whenever it is required.
- Simple interaction patterns need to be featured in the architecture so that components will perform the same things in expected ways, maintain time complexity, improve reliability, enhance modifiability, and be easy to understand. It also shows the integrity of the architecture, which is immeasurable but helps with smooth development (Kazman et al., 2000).

Characteristics

Characteristics can be well defined based on component behavior. The units of software which have defined roles and interface are referred to as components. These components build architecture on the software, and this pattern is known as service components. These components

are not just involved in building architecture they can share, communicate with, and combine with other blocks to complete a task requested by the business. They also need to be accessed by remote users, which is not an easy task; therefore, they need to be defined properly with architecture patterns. The behavior of these components defines the characteristics of the architectures, and each type of architecture has its own definition and unique topology which classifies them. The topology also defines the shape of the architecture, including the relationship between components and their actions. Based on the designated business requirements, architects can choose which topology type fits for their business with proper analysis. Overall characteristics of the components provide a better idea and impact the decision of choosing the architecture. The communication between these two architectures happens with middleware messaging in the SOA and API layer in microservices (Hutchinson, 2007).

Some rules of thumb have been discussed at this point in the paper, but some general principles can help classify good architecture. The general principles which help in evaluating an architecture are:

- Code readability/complexity
- Ease of deployment
- Scalability
- Bug resistance
- Clear data flow
- Performance/efficiency
- Test coverage

- Building/compiling speed
- Usability

Elaboration for the general principles in evaluating architecture are below:

Code readability. Few factors describe the quality of the code, such as significant naming conventions, documentation, comments, consistent code format, lack of nested callbacks, and avoiding huge class definitions.

Ease of deployment. If deploying a new feature or updating existing requirement invests more time and complexity, the architecture may require adjustments

Bugs resistance. Fixing bugs in one system can affect another method if there is more dependency between the components, which is a structural flaw.

Clear data flow. The flow of data in the application should be tracked and monitored. If architecture is allowing the capture of message sending and message consumption, it is an efficient architecture.

Performance. In large and complex applications, it requires numerous libraries, the right amount of resources, time-consuming processors, and a significant amount of data downloaded from the network which slows the application's performance. Architecture should allow the stakeholders to monitor performance, use libraries, and download different types of data in the way they prefer.

Test coverage. The major aspects which should be supported by the architecture include overall testing of the application to avoid potential errors. Avoiding these mistakes can make the environment stable and improve user experience. Architecture should allow the application to conduct testing mainly on the functionality of the application. Maximum code coverage cannot

define a well-built an app. Highlighting abstraction layers, granulation of the components, and the efficient design of the project will help in writing test scripts.

Usability. Architecture binds the developers who are imposed by an architect or a technical lead, and personnel involved in that bond should accept that contract. Architecture can be useful if the maximum number of team members can accept it (Niemela., 2002).

Service Orchestration and Choreography

There will be multiple services that need high-level coordination achieved through a mediator, which is centralized. This coordination is called service orchestration, and the mediator can be a service consumer or an integrated hub such as Mule or Spring Integration.

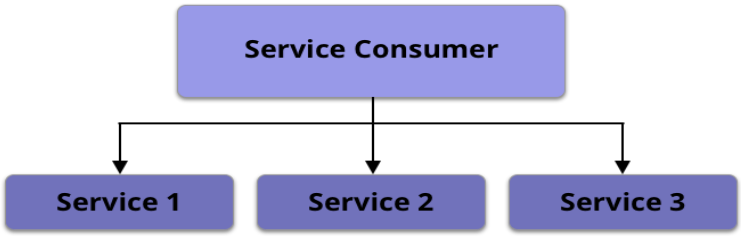

Figure 4: Service orchestration (Richards, 2016)**.**

The name is derived from the word orchestra, where all the musicians perform some music activity with the coordination of one person. In this concept, the service consumer acts as part of the orchestra and coordinates all the service functionalities to satisfy a business requirement.

The other type of process where all the services are coordinated without a centralized mediator called service choreography. The communication within the services is utilized with the conjunction of service choreography. In this pattern, one service call the other service and that service calls the next, and this chain continues until the action gets complete. And this pattern is called as service chaining. We can compare this concept with the dance choreography for better

understanding. All the performers will move their steps based on the other dancers, and there will be no coordinating persons, and they act on the previous actions (Baklouti, Sommer, & Maheo, 2017).

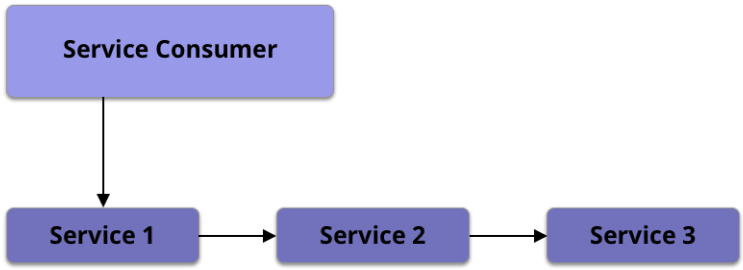

Figure 5: Service choreography (Richards, 2016)**.**

Microservices topology. In the microservices topology, there is no centralized mediator component to make the architecture follow the service choreography over the service orchestration. The two elements of architecture topology are service components and API layer. For the implementation of the product, many more components will be required such as a registry, discovery component, monitoring service, and a deployment service component. But, as per the microservices architecture taxonomy, these components are not considered as functional components but as infrastructure service components.

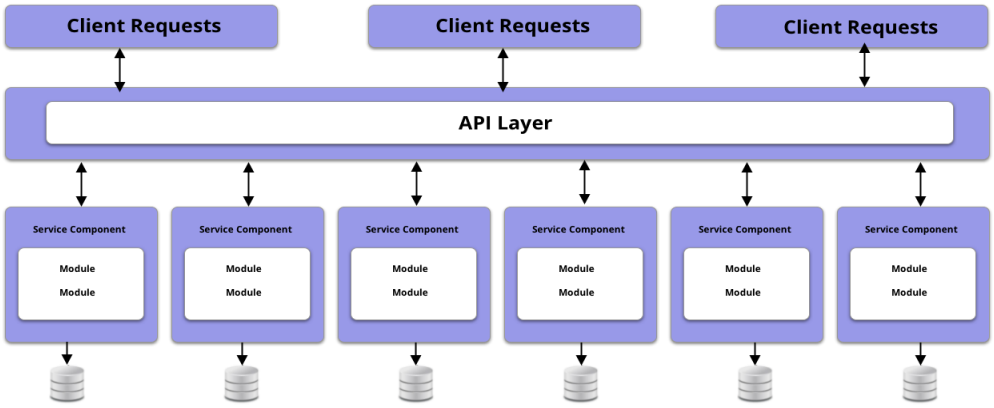

Figure 6: Microservices topology (Richards, 2016)**.**

To minimize the choreography pattern, the communications between functional and infrastructure services should be restricted or monitored. The amount of restrictions and monitoring helps with architectural decisions, as microservices are a share-a-little type of architecture which makes components loosely coupled and independent. Alternatively, if the application requires a high amount of choreography service between functional services, they are considered fine-grained services that create dependency and high-level coupling, requiring large amounts of processing time to perform a single business functionality (Richardson, 2016).

The below example illustrates a business functionality with the help of microservices topology. The order process function requires the following service components: validating the customer order, placing the order, and sending a notification message to the customer. As per the architecture design, this functionality has dependency over service, creating high-level dependency. As microservices are built based on remote access protocol, they involve high choreography which in turn impacts the response and request time.

If one service in the service chain is unavailable or not responding, for a simple business functionality will not perform well-affecting reliability and robustness of the application (Richardson, 2016). Overcoming these types of hurdles will make an increase of service choreography in microservice architecture more probable. The idea of combining fine-grained services with coarse-grained services is limited by the choreography. Fine-grained services can be treated as individual services if multiple services use the requirement of the facilities in the project. Based on the size and nature, these coarse-grained services are mapped to shared services. Following this procedure converts fine-grained services into one coarse-grained service, which in turn eliminates the usage service choreography and removes the issues raised with it.

As this reduces the availability issues, the robustness of the application can be increased. Additionally, performance can also be improved as the number of calls to the remote services also decreases (Richards & Smith, 2016).

SOA topology. In SOA topology, to build or process a single business functionality requires both service orchestration and service choreography.

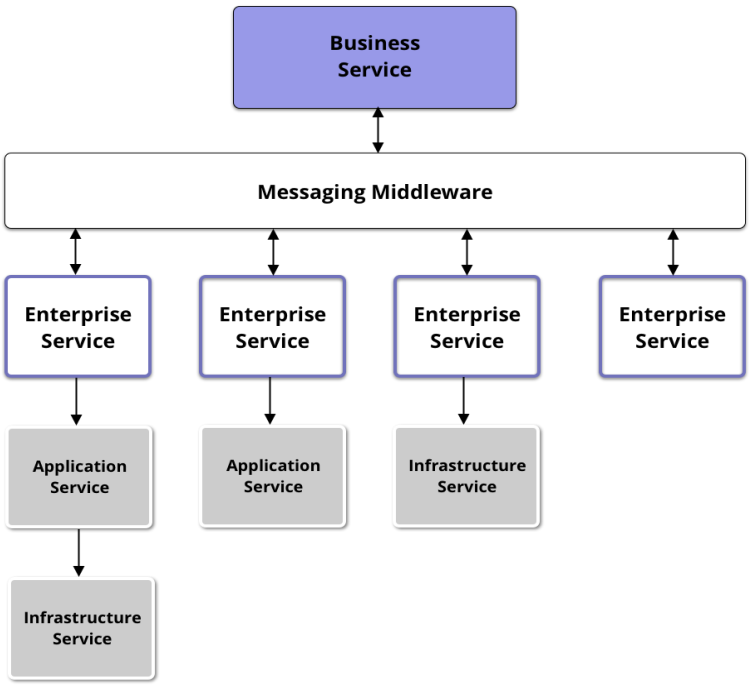

Figure 7: SOA topology (Richards & Smith, 2016).

For a single business process, SOA manages service orchestration for calling multiple business services with the help of the middleware component. After this, the service choreography calls the application or infrastructure service to satisfy the single business request. There will be multiple variations when service choreography is involved in the SOA. The flow of this topology starts by calling an enterprise service, which then calls an application service. The called application completes the business request by calling an infrastructure service. The use of

service choreography can also be eliminated if the enterprise logic is self-contained within the enterprise service and by calling application service or infrastructure service directly.

Many differences can be raised between these two architectural patterns in terms of structural characteristics involving performance issues, deployment structure, development, and testing strategies. Involvement of service orchestration and service choreography into the architectures are the reason for these pattern differences. To complete a single business request in the application, building on SOA requires more time for the process, more deployment efforts, and high maintenance in terms of development and testing, as this architecture depends on multiple business services that make SOA slower than microservices. Architects in businesses are moving towards microservices from SOA due to the issues mentioned above to achieve a streamlined and straightforward pattern (Richards & Smith, 2016).

Chapter IV: Data Presentation and Analysis

Introduction

Different architecture and analysis. There are multiple architectures in the field of software; listed below are some of the most popular architectural designs which have been selected to compare with microservices.

i. Layered architecture
ii. Event-driven architecture
iii. Space-based architecture
iv. Micro-kernel architecture

Data Presentation and Analysis

Layered architecture. This architecture is mostly used in Java EE applications. Many developers already use it without knowing that they are employing this architecture. It is also called n-tier architecture. It closely matches with the structure in which IT organizations work, which makes it a default choice to choose this architecture by many application development organizations. The main functionality of this architecture is to divide the application code into different layers and assign responsibility to each segment, with the need to provide service to the higher layers. There is no defined number of layers, but those most commonly used are:

- Presentation or UI layer
- Application layer
- Business or domain layer
- Persistence or data access layer
- Database layer

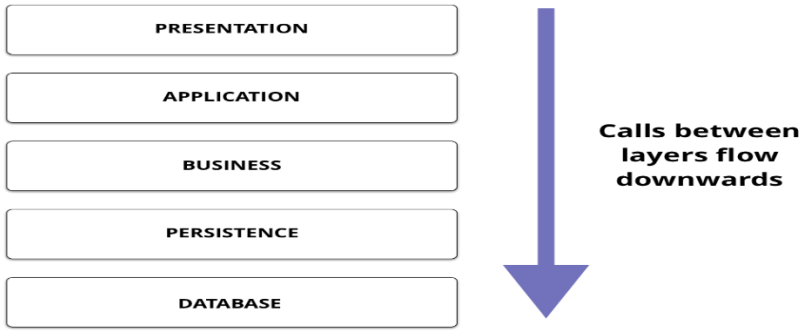

Figure 8: Layered architecture (Morlion, 2018).

The workflow of this architecture starts from the presentation layer, where the user initiates some code by performing some action on the UI (e.g., filling in data and clicking on hyperlinks). Then it will call the underlying layer, or application layer, which processes to the business layer and decides what to do with the action. It will then call the persistence layer, which stores the data into the database layer. This architecture clearly implies that the higher level layers are dependent on underlying layers and must call them to process an event. There are many variations in layered architecture, and functionality depends on the complexity of the application. Some applications do not use the application layer and instead use a caching layer, depending upon the system requirements. In some implementations, the significant layers are combined and form a single layer. For example, the active record pattern combines business and persistence layers (Morlion, 2018).

Security concerns. There are two factors which drive this layering architecture. One factor is technical and related to business capabilities, while the other factor is organizational structure. The higher layer, which is the UI layer, is driven by the security and usability of the application. Data access, data security, and privacy issues drive the persistence layer. Therefore, the UI layer is not concerned with data security and integrity, as the persistence layer

concentrates on secured data storage and its procedures by following encrypted techniques and controlling the authorizations for data access.

In the same way, the persistence layer is worried about how the data is displayed, what data type is allowed, and how long the data should be. The business layer handles the execution of specific business rules based on the user request and is concerned with the data format to be displayed on the screen rather than where data is coming from. This layer will only retrieve the data and perform business logic to it before sending the data to its higher layer. The development team subsequently divides its workforce and focuses on individual layer development. The structure will provide individual control access to each layer, which provides more security to the application. In most of the applications, only the UI layer can be accessed by external users while other layers are open to their higher level layer, which reduces the attack surface. This layering structure often exposes the business functionality of teams as the database team maintains and develops database schemas and structures, whereas designers and front-end developers work on UI requests and responses.

Pattern description. Each layer in this architecture can be treated as an individual component and are structured in horizontal layers, as shown in the above figure. Each layer performs a specific role as described, as there is no prescribed definition for many of these layers. A simple application can have only three layers, while a complex application can have more than five layers. The key feature of this architecture is a separation of concerns among its components, with the components within the layer acts according to the logic designed for that layer. Components in the business layer deal with business logic, while the persistence layer deals with persistence logic; similarly, the UI layer deals with presentation logic. This

classification of roles to each layer helps in developing effective roles and responsibility models into the architecture. It also makes development easy and removes governing issues, helps to increase test coverage, and improves handling and maintainability. The other main feature of this layer is well-defined interfaces between the components and the limitation of the component scope.

The other main type of this architecture is the closed layered architecture, where every layer in the application is treated as a closed layer. In this type of architecture, a request has to pass from every layer to reach the final layer (i.e., database layer). The request should move, and receive responses, layer by layer; as such, it takes more processing time when the user only needs to save data into the database. Unnecessary routing of requests to each layer slows down the application. The key to this concept is known as layers of isolation. The concept of isolation is that changes made to the components in one layer will not impact the other layer components. The change is limited to particular layered components and sometimes to the next layer of components when data format is concerned. If there is no closed mark on the layers and the application provides its presentation layer, direct access to persistence layer and a simple update in the SQL query in the persistence layer will impact both the business layer and the UI layer which increases the dependencies between the components of the layers so they are tightly coupled. This type of design is unsuitable for a layered architecture and building an application on this design makes updates difficult and expensive.

Considerations. Layered architecture is called solid general-purpose architecture, and is a good architecture to start developing the application when one is not sure which particular architecture should be chosen for an application. However, some concerns should be addressed

while developing an application in this architecture. The first concern is to know about architecture sinkhole anti-pattern. This concern describes how a request should flow through each layer with less or no logic applied in each layer. For example, when the user wants to retrieve the data, they will send a request to the presentation layer which passes to business layer before sending it to a particular component in the persistence layer to perform an SQL query to retrieve the data. Once the data is retrieved, it should directly stack onto the UI layer without forming any business logic or performing additional processing.

Each type of layered architectures will have some components or scenarios that fall into this sinkhole anti-pattern. The key is to find out how many components fall into this category. The 80-20 rule is a good practice to determine whether the design falls into this sinkhole anti-pattern: 20% of the components pass through this simple fall of processing and 80% involves some business logic to process the request, making the application more typical. Conversely, it can reverse simple processing applications of these numbers and cause many components to fall into the simple data flow. Achieving this flow requires some components and layers to be open, but caution should be used while opening layers. A lack of layer isolation can make changes in controls difficult. Below are some of the advantages and disadvantages of layered architecture.

Advantages.

- Most developers are hands-on with this architecture to help their teams involved in development to focus on individual layers and develop parallelly with minimal dependencies with other teams.
- Enables the development of loosely coupled systems and can achieve separation of roles.

- Easy to update and enhance individual layers, like changing the database from Oracle to SQL Server. This can be achieved just by modifying the persistence layer and can reduce testing in other layers.
- Encourages easy migration of technology stack and also provides some cohesion in terms of capabilities and responsibilities. The developers who work on the UI layer should be skilled in CSS and front-end related programming, but need not be adept in writing SQL queries.
- Achieves high security, as the persistence layer is closed to many layers but can be open to the business layer and database layer. A hacker who has access to the UI layer must break through multiple independent layers to access the data, and it is not easy to penetrate the lower layers unless an organization's interlayer security is weak.
- Different components of the application can be individually developed, tested, maintained, and deployed at different schedules, which makes a well-organized and testable application.
- Helps to configure different security levels to different components behind the firewall, as the components are deployed in different boxes. It maintains secured portions while still allowing access through a specific flow.
- Promotes high exchangeability as communication happens through well-defined protocols and interfaces, which allows swapping of implementations.
- Levels of abstractions are implemented, which enable the standardizing of tasks and interfaces. Standardized interfaces hold the change with the layer.

- High testability is achieved with the components, which are tested individually (Gupta, 2013).

Disadvantages.

- Source code can turn into a complex project if modules in the layers are not organized and roles are not well-defined.
- Less utilization of sinkhole anti-pattern can make an application very slow, even to complete a simple request.
- The main goal is to understand the layer isolation, which is difficult to grasp without having sufficient knowledge of other components.
- Though it simultaneously has technological independence, it can have tightly coupled logical components that increase dependencies across the distributed systems.
- Debugging is challenging as one request failure requires significant debugging through all layers in which data flows.
- As each layer can only communicate with the layer below or above, a simple modification in one layer results in the loss of data, which makes API calls strongly coupled and throws exceptions that can expose the code base.
- Developers are more concerned about individual layers and a simple change in one layer can affect the performance of others without modification on the dependent layer.
- As it involves testing both coupled layers, developers may try not to modify or update the components and look for an easy way to perform patchworks within the layer, which can make application inefficient.

- It requires strong governance and can impact the performance as a simple data flow requires numerous validations, which can make an application perform with low efficiency.
- Granularity and assigning roles to layers is difficult, as implementing a few layers will not be enough to handle many responsibilities and having too many layers will affect the data transmission.
- Services related to work performed by other layers might not use any other layer than its immediate layer (Gupta, 2013).

Architecture analysis. There are many factors which implement choosing architecture. Some of them are defined and rated as below.

***Ease of deployment: Rating - low*.** Implementation of the pattern impacts the deployment, particularly in large applications, as one simple change can require re-deployment of the whole application. This should be scheduled during off-hours or weekends, as this pattern does not fit continuous pipeline deployments at the risk of reducing the rating for deployment.

***Testability: Rating – high*.** Components are packed into individual layers and are independent, which makes testing the application easier. A developer can test the business layer and mock the Api's with requests and validate it on UI layer without mocking any other layers.

***Scalability: Rating – low*.** As this pattern is tightly coupled, monolithic implementations make this architecture difficult to scale. The architecture can be scaled by splitting the individual layers or replicating the whole application into multiple nodes, which propagates broad granularity and is expensive to scale.

Performance: Rating – low. Some layered architectures are considered high performance, but expecting the same level of performance in a complex application is difficult as one business request must cross multiple layers.

Ease of development: Rating – high. This is popular for most developers, as implementing a complex application becomes easier which increases the rating. Most businesses build their teams by combining different skilled people, which makes this architecture their first choice.

Overall agility: Rating – low. The application can send quick responses in a constantly changing environment. Because of the monolithic nature of the pattern, updating a component in this type of application may be difficult, which decreases the rating (Richards & Smith, 2016).

Best for:

1. New applications which have been quickly built.
2. Enterprise or business applications which mirror a traditional IT department.
3. Teams with developers who do not know other architectures.
4. Applications which require high maintainability and testability standards (Wayner, 2008).

Event-driven architecture. This is the most popular distributed asynchronous architecture, which helps in developing highly scalable applications. This pattern is adaptable for both simple applications, as well as large and complex applications. The components are single-purpose event processors which asynchronously receive and process the events, making components highly de-coupled.

The two main topologies of this architecture are a mediator and the broker. The mediator is used when multiple components are bound together to perform an event through a central mediator. The broker is used when components are chained in order without using a central mediator. As the implementation and characteristics are different in these topologies, one needs to identify which suits their application the best (Morlion, 2018).

***Mediator topology*.** This is mainly used to perform events involving multiple steps of orchestration. For example, to place a single stock trade, it requires validation of the stock, checking the compliance of that trade with the compliance rules, calculating the commission, and assigning a broker to place that trade. All these processes require some orchestration to determine the sequence of steps to be involved and also to determine which steps need to be executed parallelly and in a series.

The four important components of this architecture are event queues, event mediator, event channels, and event processors. The flow of event starts with the user sending an event to an event queue, which transmits to the request to the event mediator. This mediator receives the initial request and orchestrates that event by adding asynchronous events and sending it to multiple event channels to execute each step of the process. Event processors which receive information from event channel perform some business logic and process the event. The image below illustrates mediator topology for event-driven architecture pattern.

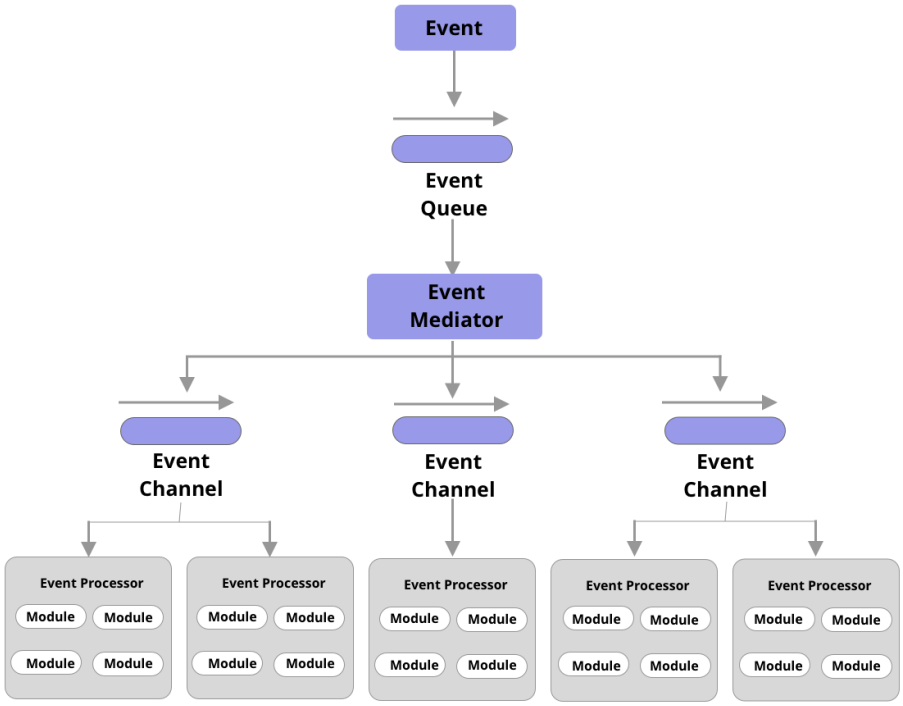

Figure 9: Mediator topology (Richards, 2016).

It is very common to have hundreds of event queues in this architecture, and this pattern will not define the implementation of these queues. It could be a simple message queue or an API endpoint.

There two types of events in this pattern: initial event, and processing event. Initial events are the original events received by the mediator, whereas processing events are generated by the mediator and passed to event processors. The mediator is only responsible for the orchestration of steps and is not involved with performing business logic. Event channels are used by the mediator to process initial event described in the steps and are then assigned to individual processors. Event processors contain business logic and process the events. They are self-contained and highly independent, and only perform the assigned task in the application. The

granularity can range from fine-grained to coarse-grained. Passing the events to correct processors is a challenge when building a successful application.

***Broker topology*.** This topology will not have a mediator, and the event flows to multiple processors like a chain with the help of a lightweight message broker. It is more advantageous if it involves simple event processing flow. The two main components in this pattern are the broker component and the event processor component. The broker component is federated and contains event channels used within the event flow. The image below illustrates broker topology.

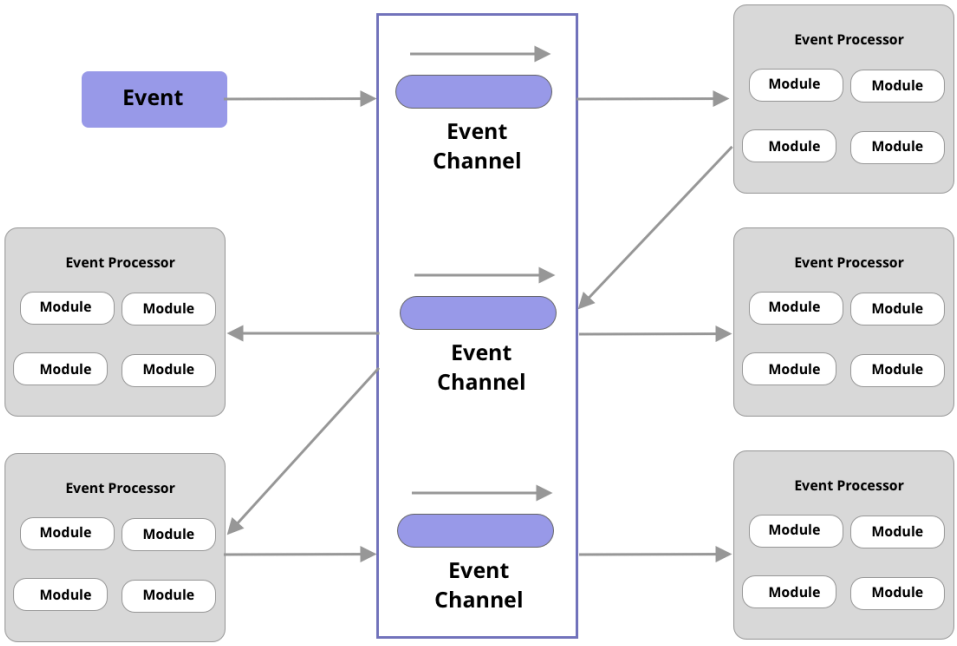

Figure 10: Broker topology (Richards, 2016).

There is no mediator to process the initial event. Here, each event processor is responsible for processing the event, publishing them to other events, and sending back a message that the event processed. There might be some processors that handle these events and the published message is not consumed by any other event, which may leave the application open for future updates.

An example of this pattern is to update an address for an insured person. The initial event is received by the consumer process component. It updates the data and sends out a message containing the updated address. The claims processor and quote processor are interested in receiving this message. The quote processor consumes this event and calculates the new rates based on the new location and sends out a message, whereas the claims processor consumes the event, checks the pending claims, processes it with the new address, and sends out a message. These messages, received by another process, handle the event and notify others until no more processing is required. This pattern is all about breaking the business logic processed eventually by the event (Richards & Smith, 2016).

***Considerations*.** Broker topology is a bit complex to implementation due to its asynchronous distributed nature and addressing various distributed issues, such as availability of remote processes, lack of responsiveness, reconnecting issues when broker or mediator fails. One of the main considerations is the lack of atomic transactions for a simple business process. This increases the difficulty of maintaining a transactional unit across the system as event processors are tightly decoupled. The architecture must then decide the granularity of the event processors based on the dependency level of the processors, which may not be suitable for the applications as they require undivided work and must process an event in a single transaction. Difficult aspects of this architecture include developing, maintaining, and governing event processors. Each processor has different contracts in accepting data formats and minimizing the use of multiple data formats in different processors (Morlion, 2018).

***Advantages*.**

- Loosely coupled nature allows easy communication without disturbing the layers.

- High flexibility, as programmers have control to place the code and triggers at the start of the event.
- Allows more interactive programs and programming, making the application layout simple.
- Event processors help decide the business logic, which provides insight (Kahadawa, 2006).

***Disadvantages*.**

- Events traverse in different directions, making traceability difficult.
- High broadcast of traffic as a simple change in the event is pushed to all the consumer events regardless of necessity.
- Event context is duplicated for each observer in a distributed system, reducing memory scalability.
- Classes in this pattern are not reusable and restrict portability to other operating systems. Event loops consume a lot of processing which impacts performance (Kahadawa, 2006).

Architecture analysis. Below are the ratings for event-driven architecture:

***Ease of deployment: Rating – high*.** Deployment of this pattern is easy due to its decoupling nature. Broker topology is easier to deploy when compared to mediator topology, as the mediator is tightly coupled compared to broker topology. A change in event processor requires a change in a mediator, which involves both in deployment.

Testability: Rating – low. Individual testing is not difficult, but performing testing on the overall application requires some testing tools or testing clients to generate different event types. It increases the complexity in testing due to the asynchronous nature of this architecture.

Scalability: Rating – high. As this pattern is highly independent and decoupled, it is easy to achieve scalability in the application. It allows fine-grained scalability, as each event processors is scaled individually.

Performance: Rating – high. Due to high traffic involved in this pattern, it is common to have low performance. However, the asynchronous nature of this architecture increases performance. The nature of this pattern allows parallel operations and dequeues the messages, which rate this pattern as high.

Ease of development: Rating – low. Due to this asynchronous nature, developing this pattern is difficult. This development bit is made complex by contract creation and involvement of advanced techniques handling error and exceptions, as well as implementing logic for unresponsive and unused messages.

Overall agility: Rating – high. The application can send quick responses in a constantly changing environment. As event processors are independent and used for a single purpose, change in a single component is isolated to itself and gives more freedom when updating components (Richards & Smith, 2016).

Best for:

1. GUI-based applications, which have made this pattern more popular.
2. New business services, invoking listeners dynamically and transparently.
3. Asynchronous systems with the asynchronous data flow.

4. Applications with individual data blocks, which tend to communicate with few or single components (Wayner, 2008).

Space-based architecture. Most web-based applications are built around a database and perform well as long as the database keeps up the load balance. Nevertheless, the database cannot log all the transactions when usage is high, causing the application to fail. To handle functional problems, space-based architecture has been introduced to divide processing and storage into multiple servers. The data is distributed across the nodes and is assigned responsibilities. This pattern is also called cloud architecture, where processes and data are split up to multiple nodes. It supports the application, which involves in deleting the database. Using RAM to store information makes the system highly performable, and distributing memory and processors can simplify basic tasks. Computations must be spread out across the data set; for example, statistical analyses must be split into sub-jobs and need to be aggregated when the process is finished.

The two primary components of this architecture are the processing unit and the virtualized middleware. The processing unit consists of a unit (or portions of units) of application components. Including web-based components and business logic components. This architecture varies for different applications. Smaller web-based applications can be deployed into a single processing unit while complex applications need multiple processing units, which are organized based on the functional requirements. This component consists of application modules, in-memory data grids, and persistence stores to handle failures. The virtualized middleware component handles communications and internal house-keeping. It also handles requests, synchronization of data, and various controls. The components inside this block are the message

grid, data grid, processing grid, and deployment manager. The image below illustrates more information about this architecture (Morlion, 2018).

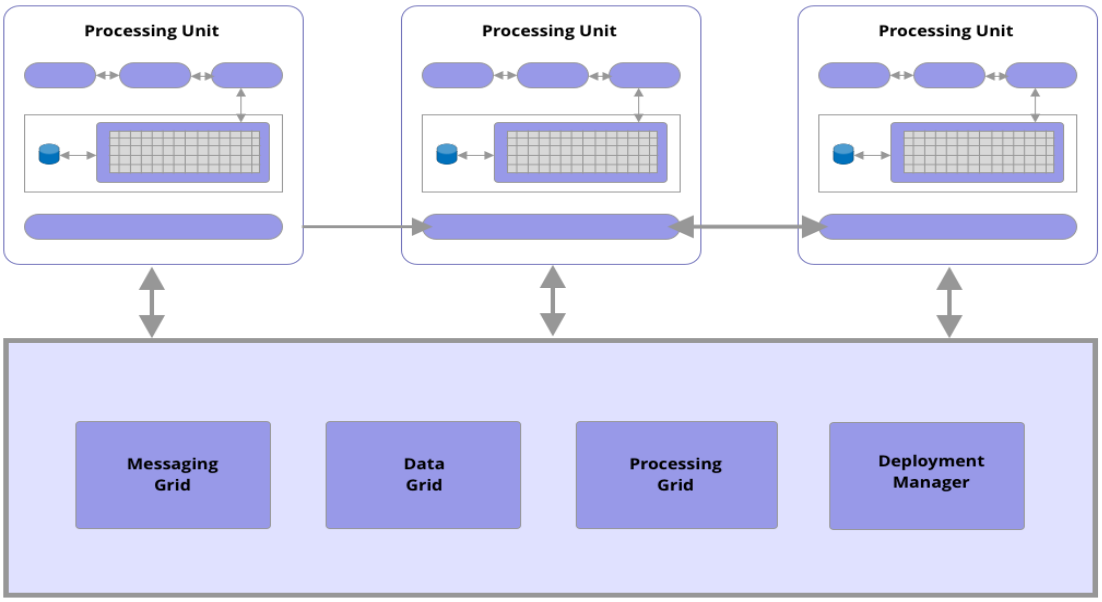

Figure 11: Scape-based architecture (Richards, 2016).

The main functionality of this pattern lies in virtualized middleware and in-memory data grids. It is the most essential and foremost controller of this architecture and manages requests, data replications, managing sessions, request processing distribution, and unit deployments.

The first component is the message grid, which handles input requests, detects the active processing components, forwards requests to those units, and manages sessions information. The complexity range from either a simple or complex algorithm and tracks the information regarding which unit processed which request. The other component is the data-grid, which is the most important and crucial component. It communicates with a data replication engine to manage the replications in the data when an update request occurs. It is handled in parallel and asynchronously to perform quick actions (Mordinyi, 2010).

The other component is the processing grid, which is an optional component in virtual middleware that manages distributed requests where multiple processes handle portions of the request. When a request requires multiple processing units, the processing grid mediates and orchestrates between the multiple processing units. The last component is the deployment manager, which handles and initiating and shutting down processing units based on their load. The major responsibilities of this component are to monitor the response time and data loads, assign processing units when the load is high, and shut down the processors when the load decreases (Mordinyi, 2010).

Considerations. This architecture is very complex and expensive to implement and is suitable for smaller web applications with a defined load. It is not suitable for traditional RDBMS applications, which have large operational data. This pattern does not require centralized data storage, which can be performed on initial in-memory data loads and persists data updates asynchronously. The widely used transactional data can be divided from non-used and inactive data, which reduces the memory data grid. The components of the architecture require no deployment on the cloud-based servers or platform based servers. There many third-party products like Gemfire, GigaSpaces, and Oracle Coherence, which supports this architecture which makes it easily adaptable. The main challenge for the architect when choosing this pattern is to clearly specify goals and make use of every product to keep the application cost-effective (Mordinyi, 2010).

***Advantages*.**

- Useful for applications which handle simple and large data processes.
- Ease of implementation, due to third-party providers.

***Disadvantages*.**

- With RAM DB, transaction monitoring is complex (Mordinyi, 2010).
- Testing is complex, as the generation of a high load is challenging.

Architecture analysis. Below are the analysis and ratings for space-based architecture.

***Ease of deployment: Rating – high*.** Deployment is simple, though the components are not decoupled but instead are dynamic and sophisticated cloud-based tools. It allows the code to be pushed to the servers and reduces the complexity in deploying the application.

***Testability: Rating – low*.** Building high loads of data to test the application is more challenging. To create this type of environment to test the application is more expensive and time-consuming. Testing the scalability of the application also adds difficulty.

***Scalability: Rating – high*.** Scalability is high due to low and negligible dependency on centric data, simplifying the calculation of scaling the application.

***Performance: Rating – high*.** In-caching mechanisms and in-memory data access allow this pattern to be classified as a high-performance architecture.

***Ease of development: Rating – low*.** The components of the in-memory data grid and sophisticated caching products are complex to develop. As the tools and the products used to develop are not so popular, many developers may not be familiar with these technologies.

***Overall agility: Rating – high*.** The application can send quick responses in a constantly changing environment. Processing units and servers involved in handling the requests can manage high traffic, give responses quickly, and decrease the server load to help even a simple request to process quickly (Richards, 2016).

Best for:

1. Small applications with a pattern dynamic in nature.
2. Social networks, search engines, and applications handling low-value and insensitive data transactions like bidding applications (Wayner, 2008).

Micro-Kernel Architecture

The most popular architecture used for developing a product-based application is micro-kernel architecture. Product-based is a type of application which can be downloaded and updated in version and used on local machines, which are also known as third-party applications. It can be applied to fully developed applications, plugins for browsers, or OS-based applications. Micro-kernel provides extensibility, decides core functionality, and adds other application functionalities through plugins. The most famous tool, Eclipse, is built based on this architecture and reads, opens, edits, and annotates files. With add-on plugins, it will provide more functionalities to the files, and with one click it will compile the files in the compiler.

There are some other tools which have other language compilers beyond Java which are added as a plugin to these applications. Add-on features on top of the layers are also called an extensional approach, and this pattern is also called a plug-in architecture (Ticki, 1998).

Pattern description. The two main components of this architecture are the core system and plug-in modules. Application logic distributed among these components provides high flexibility, isolation of features, extensibility, and customized processing logic for plugin-components. The image below illustrates micro-kernel architecture.

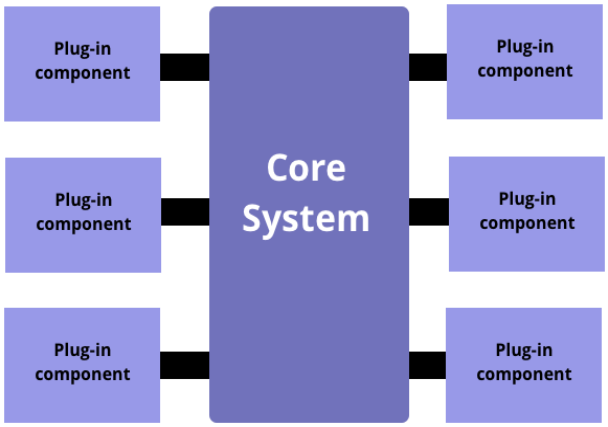

Figure 12: Micro-Kernel architecture (Richards, 2016).

The core system in the pattern has minimal functionality and major responsibility to keep the application up and running. Typically, this component will have general business logic and some special features to handle special rules and complex conditional processing.

The plug-in components attached to the core system are independent and contain specialized processing, additional features, and additional business logic which can act as an extension to the core system. Completely independent plugin components can be developed or made to work for one another, but minimizing the communication will avoid dependency issues. Core systems are connected to a registry plugin containing all information about other plugins, like contracts or protocols. When a request hits the core system, it will get information from the registry and send the message to that particular plugin. The communication can happen via pen service gateway initiative (OSGi), message passing, API, or direct point-to-point binding (Bonilla, 2015).

Considerations. The great feature of this architecture is that it can be embedded or used as a part of other patterns. If this feature cannot be implemented in the existing application developed in microservices, this architecture can provide a solution. The microservices pattern

can embed this feature into its application with the help of the micro-kernel, giving more independence while updating the features. First, the core functionality can be developed, later based on the requirements plug-in components can be implemented without making significant changes to the core functionality. For product-based applications, micro-kernel would be the best choice if one wants to release features over time and have control over the availability of the additional features based on the user type. Over time, if this pattern is not supporting the requirements, it can always be refactored to more suitable patterns.

Advantages.

- Separation of services makes this pattern reliable; a failure in one service will not crash the whole system.
- Different features set in different modules make this pattern more maintainable and testable, as loading or updating individual servers and performing testing on them will not impact the other modules.
- Passing messages are independent, eliminating the need for whole system rebooting.
- Provides easy and reliable integration with other third-party modules.

Disadvantages.

- Memory footprint is large, which causes potential performance loss.
- Message-passing bugs require more debugging, and managing the process is complex.
- Choosing the granularity is challenging in the early stages and hard to modify in later stages (Bonilla, 2015).

Security concerns. This pattern is more secured when compared to monolithic patterns. The direct consequence with the principle of least privilege is the minimality principle of micro-kernels, which means extended components will be restricted and must be privileged to complete a task which is assigned and requested. Critical bugs in a monolithic pattern are handled easily in this pattern (Eeles, 2004).

Architecture analysis. Below are the analysis and ratings for micro-kernel architecture.

***Ease of deployment: Rating – high*.** This completely depends on the implementation strategy. Plug-in components can be dynamically added to the core system during run-time, which reduces downtime during deployments.

***Testability: Rating – high*.** Plug-in is mocked and tested in an isolated environment. Core system can mock the plugin to create an environment to test the particular plug-in without making significant changes.

***Scalability: Rating – low*.** The micro-kernel pattern is implemented based on products, which are smaller in size, treated as a single unit, and therefore difficult to scale. Implementation strategies for the plug-in can be scaled, but the overall pattern can be rated low in terms of scaling.

***Performance: Rating – high*.** The micro-kernel pattern is avoided for developing high performance or complex applications, but products developed in this pattern can perform well as required features can be customized and pipelined.

***Ease of development: Rating – low*.** The required pre-defined design and contract governance make this architecture complicated to implement. Contract versioning, maintaining

internal plug-in registry, choosing granularity, and wide ranges of choosing plug-in make it difficult to develop an application using this pattern.

***Overall agility: Rating – high*.** Changes can be made to the individual plugin and be isolated, which makes them loosely coupled modules. The changes can, therefore, be implemented quickly and make the application more stable regardless of some changes in the environment. This pattern is robust (Richards & Smith, 2016).

***Best for*:**

1. Applications which expect no memory leaks, no buffer overflows or handling exceptions.
2. Tools which have a wide range of user types.
3. Applications which have a defined the division line between basic functionalities and request-based functionalities.
4. Products which have defined sets of routines and dynamic sets of rules for feature updates (Wayner, 2008).

Architectures / Factors	Layered	Event-Driven	Space-Based	Micro-Kernel	Microservices
Ease of Deployment: Rating-	Low	High	High	High	High
Testability: Rating-	High	Low	Low	High	High
Scalability: Rating-	Low	High	High	Low	High
Performance: Rating-	Low	High	High	High	Low
Ease of Development: Rating-	High	Low	Low	Low	High
Overall agility: Rating-	Low	High	High	High	High

Figure 13: Overall analysis of architectures (Richardson, 2016).

Designing microservices architecture. In software architecture, the main concerns in building the application based on microservices are availability and scalability. The lead issues that need to be considered while building the architecture are:

1. For deployment: focusing on cloud and container technologies.
2. For migration: focusing on modernizing legacy systems.
3. For sustainability: focusing on software evolution and maintenance.

The resources provided by the infrastructures have elasticized leverages when microservices are involved, which lead architects to focus on container and cloud technologies. Security and vendor lock-in are the two major barriers to adopting cloud technologies. Vendor lock-in is considered as a major concern for considering MSA, as it decreases the portability of applications. To avoid this constraint in infrastructure services, the architecture needs to be agnostic in cloud technologies. There are no proper focuses on the properties of quality

assurance and their impacts. The systematic mapping study highlights this perspective. The system-level quality properties such as availability, reliability, performance issues, maintenance of the application, testing factors, and security concerns are impacted and have a direct relationship with the architecture of microservices.

Estimation of the level of services granularity is also a major issue in designing the architecture and can be considered as trade-off concerns when it comes to the size and number of the services and quality satisfaction between local and global properties. The quality properties of SOA on the system also affects the classification of MSA quality properties in both positive and negative ways while migrating the architecture. To identify the potential advantages of MS architecture, more focus must be placed on investigating technical reasons, tactics followed to design, analyzing architecture, and the quality analysis properties along with their trade-offs (Jamshidi, 2016).

There are currently very few architecture languages that help describe and design MS architecture. AL's is the only promising language used for describing service-based architectures. SOAML, SOMA, SOADL, and Stratus ML are considered as relevant modeling languages for SOA, which was adopted in 2009. The modeling services in UML profiles are implemented using SOAML for system and business levels, service contracts, and interfaces. For distributed environments, SOAML extending from UML2 supports explicit modeling service. This type of service uses most of the advanced UML tools.

The other modeling technique which provides service identification, specification, and realization is Service-Oriented Modelling and Architecture (SOMA), which was introduced by IBM (Arsanjani et al., 2008). Modeling services like interfaces, behavior, semantics, and quality

are taken care of by SOADL. This modeling language also provides analysis and modeling mechanisms for the dynamic and evolving architectures. It is supportive of architecture-based composition services. Run-time behavior, the configuration of the application, and the services that must be defined in the modeling framework for cloud applications are all handled by Stratus ML. It is achieved by setting up adopted rules and estimating the cost for diverse cloud platforms and configurations. A detailed analysis is required to evaluate whether these languages are suitable for modeling and analyzing microservices architecture (Jia, Ying, Cao., & Xie, 2007).

The major concern for adopting MSA is moving the existing systems or services to microservices. Difficulties in migration include a lack of specific documentation of the techniques used for safe migration. If the application migrated, the identification of any positive or negative impacts becomes difficult. It is considered a non-trivial task and is multi-dimensional for adopting cloud architectures in the migration process. It is the main challenge for the migration and many companies desire a cloud as their target platform when migrating or simply researching solutions for cloud computing issues. Migration of web applications to the cloud is one of the proposed migration techniques a business can adopt when moving toward microservices, as research on microservices is in its early stages. As an already mentioned adaption to cloud computing is to take full advantage of cloud platforms, architects consider this a non-trivial task.

A new approach proposed for a model-driven methodology called MODACLOUDS allows and adopts a cloud-agnostic way to design software systems, with the development being model-driven. This development supports the process of addressing the system into multiple clouds with the help of model transformation technology. There will be considerable influences

when technology adopts the cloud, and this can be analyzed by MODACLOUDS (Danilo et al., 2012). It also performs risk analyses when selecting cloud providers and evaluating multiple clouds which run under the same system. Under the execution of the application, the runtime environment must be set up.

For situational methods in engineering, migration patterns for microservices are defined in an initial repository building the basic ground for achieving the goal of reusing the knowledge perspective. Multiple challenges exist for procedures that split large applications into small and independent services. These are considered incremental and iterative procedures in the process of migration. It acts as a service pattern on the data-driven concept when approached as SMART or Entice, which are basic service boundaries impacting the decomposition process. The methods are still obscure in estimating both the qualitative and quantitative impact of migration. Though there are many types of ongoing research into the migration process, it is still considered a risk-involved process (Mohammadi & Mukhtar, 2013).

Deployment Factors

The deployment procedure for monolithic architecture involves executing the large and single or multiple identical copies of an application on servers. If identical applications are running, every instance of the application executes on individual servers. Though it is considered a simple deployment procedure when compared to microservices, architecture-based application deployment is complicated. In microservices, there are multiple services developed in different languages and respective frameworks, which are treated as mini-applications for their specific deployment strategies, involved resources, scaling of the application, and monitoring requirements. The main challenge in this type of architecture deployment is calling services, and

instances of the services need allocated processing space in the CPU, memory allocations, and I/O resources. There will be multiple instances for individual services. Apart from this complexity, other challenges like fast processing, reliability, and cost-effectiveness are present.

There are multiple microservices patterns that can be defined, some of which include:

a) Multiple service instances per host pattern;

b) Service instance per host pattern;

c) Service instance per container pattern;

d) Serverless deployment.

Multiple service instances per host pattern. Using multiple service instances per host is one of the strategies to deploy a microservices pattern. It is the traditional way of deployment where multiple service instances are run on provision hosts, which can be remote or physical hosts. Multiple hosts can be used; every service instance runs on a well-known port on the host, with these hosts considered as pet hosts.

There are two types of variants for this pattern; one is in a process or multiple processes and executes multiple instances of services (e.g., the same Apache server executing multiple Java-based web applications; one OSGi container running multiple bundles of OSGi). The other variant, as opposed to the first, runs each instance of services in a single processor process group. For example, the Java service instance on the Apache Tomcat server needs to be deployed as a web application. There are many advantages and disadvantages to using this type of pattern.

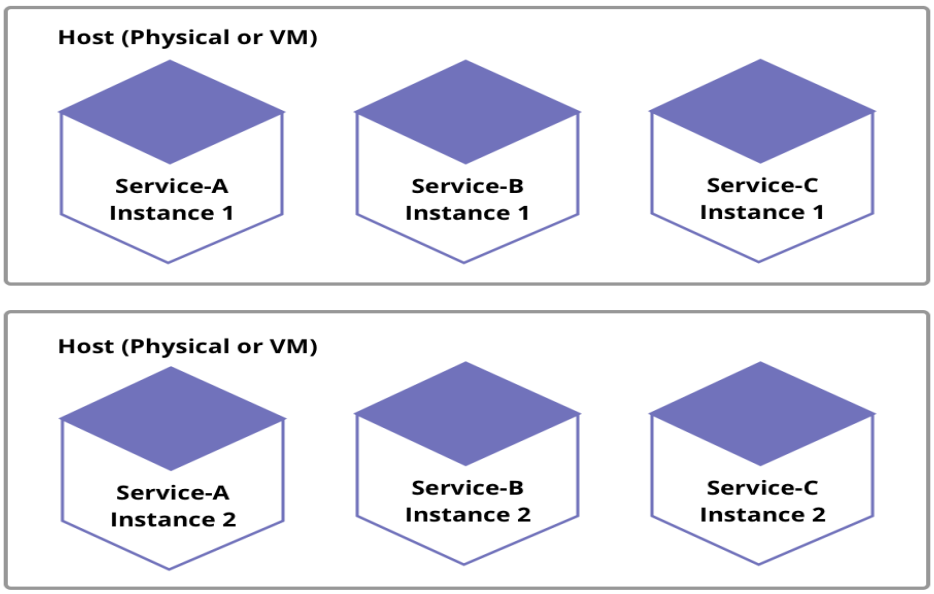

Figure 14: Multiple service instances per host pattern (Richardson & Smith, 2016).

With this pattern, the relative efficiency of resource usage is obtained, and multiple web-based applications can be run on a single tomcat server or JVM. Multiple instances of services running on a process make resource usage efficient. The other advantage of this pattern is a decrease in memory usage, as the source code copied on the network will be very small. A service can be copied to the host and start the process of execution, and the deployment of an instance will be simple and fast when compared to other patterns. For example, JAR or WAR files must be copied to run Java service instances (Kong, n.d.). Initiating the service is swift due to lesser overhead, and the process can be started when the service has its process. The multiple instances running in a process can be dynamically deployed into the container, or the container can be set to restart.

Apart from the advantages, there some disadvantages with this pattern. A major disadvantage is that the isolation of service instances will be minimal if the only instance of the services runs on an individual process, resulting in little to no isolation. The resources used by

each instance cannot be controlled and the resource utilization can be monitored. There is a chance of consumption of all CPU processing or the memory of the system if one service instance behaves abnormally. One instance of this kind can affect services that share a process between them. Sometimes, this affects the monitoring of the usage of resources. As the services are written in multiple languages and frameworks, they require strong communication between the developers and the deployment teams regarding the details of how these services function (Richardson, 2016).

Service instance per host pattern (VMs). This is another pattern to deploy microservices where only one service instance runs on its host to maintain isolation. The two major specializations of this pattern include running service instances on a single virtual machine (VM) and an instance in a single container. In service instance per host pattern, the single service instance is considered as a package for a VM image. There are multiple tools available in the market which can be used to build a VM and configure a server to be used for continuous integration. Animator is one of the procedures to package services as EC2 AMI, but it does not support many virtualization techniques such as VMware or Digital Ocean. To avoid these issues, we can use another method called Packer, which creates VM images automatically.

Another method to create minimal VM images by packaging a Java application is Box Fuse. This procedure allows fast building, booting, and has limited attack surface, which in turn increases security. Bakery, which is introduced by Cloud Native for creating EC2 AMIs, is a process that can be invoked once the microservices pass a test of configuring servers which have continuous integration. With this method, the time consumption for creating AMI can be avoided

as a bakery takes the responsibility of packing services to AMI (Taibi, Lenarduzzi, & Pahl, 2018).

The major advantage of this type of pattern is complete isolation. It is set to run a service instance with a fixed amount of usage of CPU and memory; more space from other services cannot be acquired. It also supports load balancing, autoscaling features provided by the AWS cloud and leveraging mature cloud infrastructure when VMs are deployed. Deployment of service handles the VM's management API, which makes the deployment simple and reliable (Richardson & Smith, 2016).

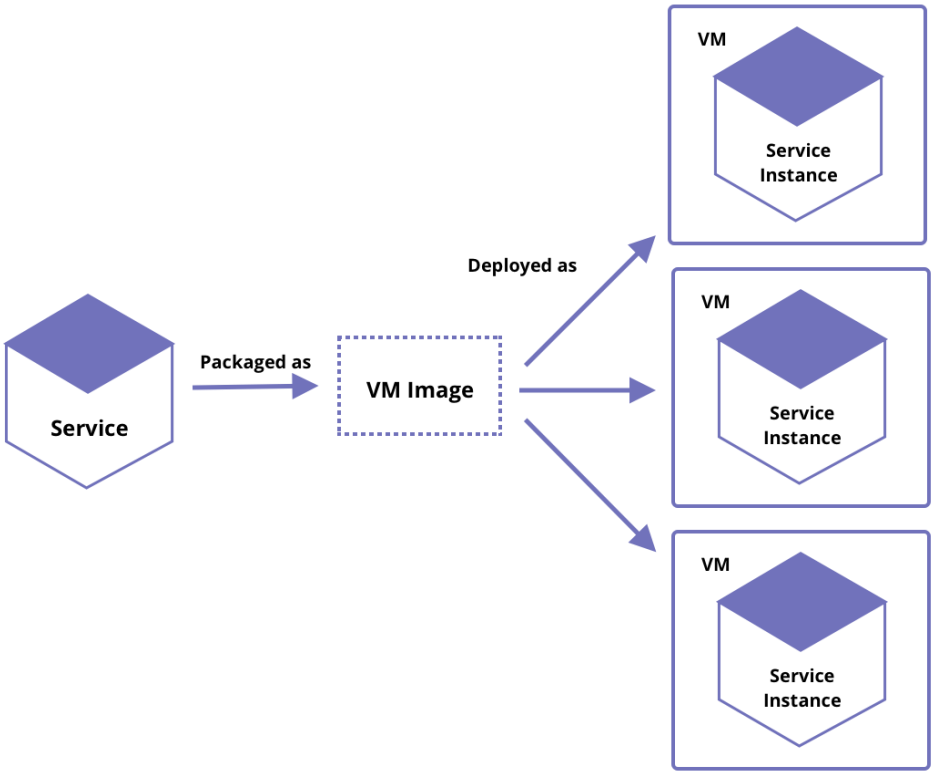

Figure 15: Service instance per host pattern (VMs) (Chris Richardson, 2016)

There are some drawbacks with this type of pattern. As Service Instance per host pattern are backward when compared to service instances, including the OS, efficient resource

utilization is limited. Sometimes VMs might not be fully utilized, as they come fixed in size regardless of their usability in infrastructure as a service. Though it has autoscaling, it is not quick in adopting changes. This drawback affects the deployment costs, as the VMs need to be altered as per the demand. Using this pattern, more focus must be put on on heavy lifting, which is necessary but affects the main business functionality by consuming time.

Service instance per container pattern. This is another type of pattern, where instances of each service run on their containers. These containers are considered as VMs at the OS level. This container allows multiple processes to run on each container and each of these processes has their port namespace and root path system. The main advantage of this pattern is that the containers' CPU process and memory can be monitored and controlled. Docker and Solaris are container technologies, and some implementations have input and output limiting as well. In this pattern, container images are used for packing services. To run the service, dependent applications and libraries are necessary, and these are stored in the image. These images can also be called file system images, and some consist of whole Linux path file systems. Java runtime, Apache servers, and fully compiled Java application are required in the image of a container to run and deploy Java service. On every single host, multiple containers can be run by packing the services into the container images. Kubernetes and Marathon are the cluster managers helping to manage containers. Managing means assigning places to containers based on the availability and requirement of resources on the host.

Like VMs, there are many advantages with this type of pattern, like isolating instances of one service from other and high-level monitorization of resource consumption by containers. This pattern also encapsulates technologies used for services by containers and utilizes APIs for

managing these containers. Services are also managed, but the main difference between containers and VMs is the use of lightweight technology for the creation of images. The images of containers are very fast in action in terms of processing, building and restarting. Whenever the container is started, it directly invokes the service to execute as it does not have a lengthy OS booting time (Lago, Malavotta, Muccini, Pellicione, & Tong, 2015).

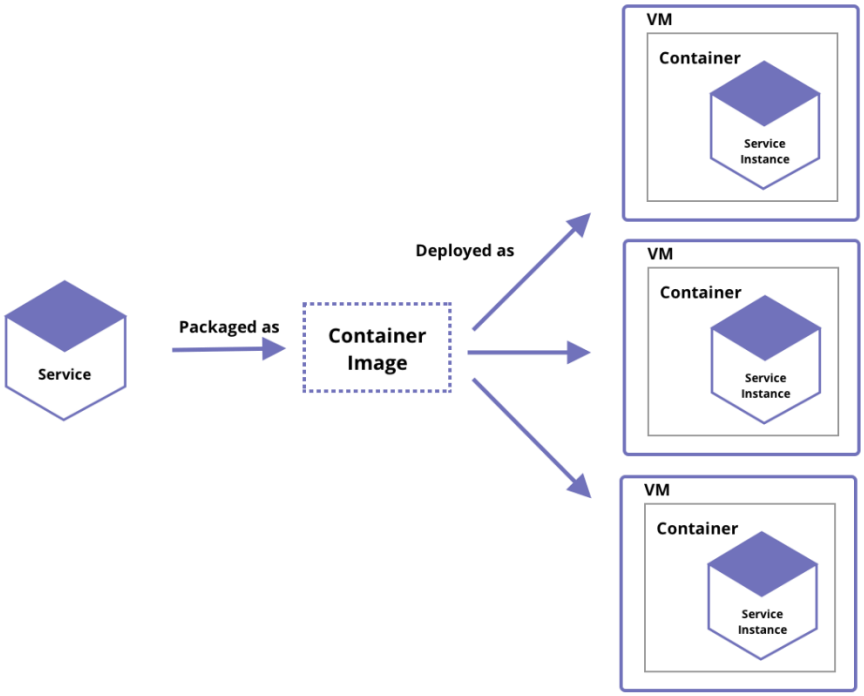

Figure 16: Service instance per container pattern (Richardson & Smith, 2016).

Containers are new and swiftly developing technology, but it is not as stable as a VM's infrastructure. This fact is considered one of the disadvantages of container infrastructure.

In their process, the kernels of the OS hosting them share with other containers, making them less secure than VMs. If hosted container solutions are not being used in infrastructure, the container images must be controlled and managed, and sometimes they cause heavy lifting. They are also not cost-effective when compared to VMs and cannot rely completely on container

infrastructure, as VMs may be required to handle spikes. Both infrastructures have some differences in their efficiency and advantages; choosing in between these two is dependent on an application's requirements (Toffetti, Brunner, Blochlinger, Dudouet, & Edmonds, 2015).

Serverless deployment. This is another new infrastructure that is booming in the industry and supports services developed in major platforms like Java, Node.js, and Python. An example of this infrastructure is AWS Lambda technology, which accepts ZIP files and metadata of the methods processing the service request. In this technology, AWS will select and run the required instances only to process the request, which is cost-effective and memory-efficient. This technology avoids the major aspects of VMs, containers, or servers. With the help of AWS services, Lambda functions process the request. Within a Lambda function, a method to receive the data can be created, stored in the required database fields, and pushed for streaming or processing. They are also known as third-party web services. Lambda functions can be invoked with the following methods (McGrath & Brenner, 2017):

- Sending a request to a web service.
- A request generated by invoking AWS services automatically invokes Lambda to send a response.
- In response to an HTTP request from the clients through API Gateway.
- Invoking periodically based on requirement or schedules.

This technology allows more freedom to focus on development, as it takes responsibility for infrastructure. It allows this pattern to efficiently deploy microservices. Despite having many advantages, it has some drawbacks as well. It is not supportive of services that take a long time to process. As it runs individual instances for every request, there are no defined states for

services to process effectively. It also has limitations in using language to develop the services, and these services need to be quick or the process might terminate (Castro, Ishakian, Muthusamy, & Siominski, 2017).

Chapter V: Recommendations and Conclusion

Recommendations

Integration of microservices. To maintain the autonomy of microservices, choosing effective integration technology is important. The feature of autonomy gives the freedom of updating and developing the application without dependency. Protocol for building the communication between the microservices and application clients needs to be decided during earlier stages when building an application as a set of microservices. Every service in this architecture has its independent set of fine-grained endpoints. Consider a shopping mobile application with the following microservices:

a) Shopping cart service: maintains the number of items

b) Shipping service: maintains shipping information.

c) Inventory service: for inventory data

d) Recommendation service: service to provide recommendations to users

e) Order service: maintains an order history

f) Review service: manages customer reviews

g) Catalog service: maintains product catalogs

If a user wants to obtain the details of the product in these types of applications, they need to call the REST service. In regular monolithic applications, that service is called "api.shopping.com/productData/productNum." Once the call is instantiated, the load balancer takes the request and sends it to identical instances, then that application request enquires with the database and sends the response back to the user. But in microservices, if the client sends this request, the data which needs to be retrieved is linked with multiple microservices like a catalog,

order, inventory, etc. The request needs to be mapped to the services and requires a response (Newman, 2015). There are many ways to implement the integration of microservices; two popular techniques include direct client-to microservice communication and using an API gateway.

***Direct client-to microservice communication*.** This integration process creates a direct communication between the client and microservice. As every service in microservice architecture has their respective endpoints, a request like using the URL https://requestedserviceApi.shop/request/ links the request to the microservice load balancer, which routes to matched instances and collects the data. As the data is related to multiple services, it has to request its related services. There are some drawbacks with this approach, as there might be differences with the end API and the request from the client. Another issue is that it must make a request for all the related APIs of services, which makes it complex if the application has several microservices. Products like Netflix or Amazon have hundreds of microservices, and it must make several requests for a single page to be displayed, which makes the application complicated. The pattern will become even more inefficient if the client requests the LAN or a mobile network.

Direct communication between the client's request and microservices can lead to security issues if protocols are not secure oriented. Some services may use different protocols other than HTTP or web socket that are not approved by the firewall. This problem arises if the client's request is outside the firewall; this is a non-issue if the request is from internal sources. It also limits the refactoring of microservices. During the development process, some of the microservices need to be changed, updated or merged with other services. If there is a direct

connection with the client, the refactoring of services will become a difficult issue. With the limitations and problems, it makes this approach less likely to be used and pushes developers to migrate towards another approach (Richardson & Smith, 2016).

***Using API gateway*.** This is another option for integration of microservices over the problems of the above approach. API gateway is nothing but a server and is treated as a single point of entrance for the application. It is the same as the Façade procedure, which is used in object-oriented designs. API gateway packages both the application and architecture while providing a single API to the client requests. This service also takes additional responsibilities like monitoring requests, cache, authorization, authentication, and request and response handling. Once this API receives a request from the client, it routes the request, composes them, and takes care of translating different type of protocols. API gateway internally calls the required microservices and collects responses from them. If there are any protocol differences, it has the translation capabilities to handle friendly protocols like HTTP and internal protocols. It also creates a customized API for every client request and makes them available for end clients. With that customized API, the clients can get the results for their requests by calling the end-point API.

To obtain the details of the product, API gateway creates a URL like "shop.com/productlist?productnum=abc", which helps the client to retrieve data with a single request. This API is provided by gathering the results from all the required microservices and this is handled internally by API gateway (Newman, 2015).

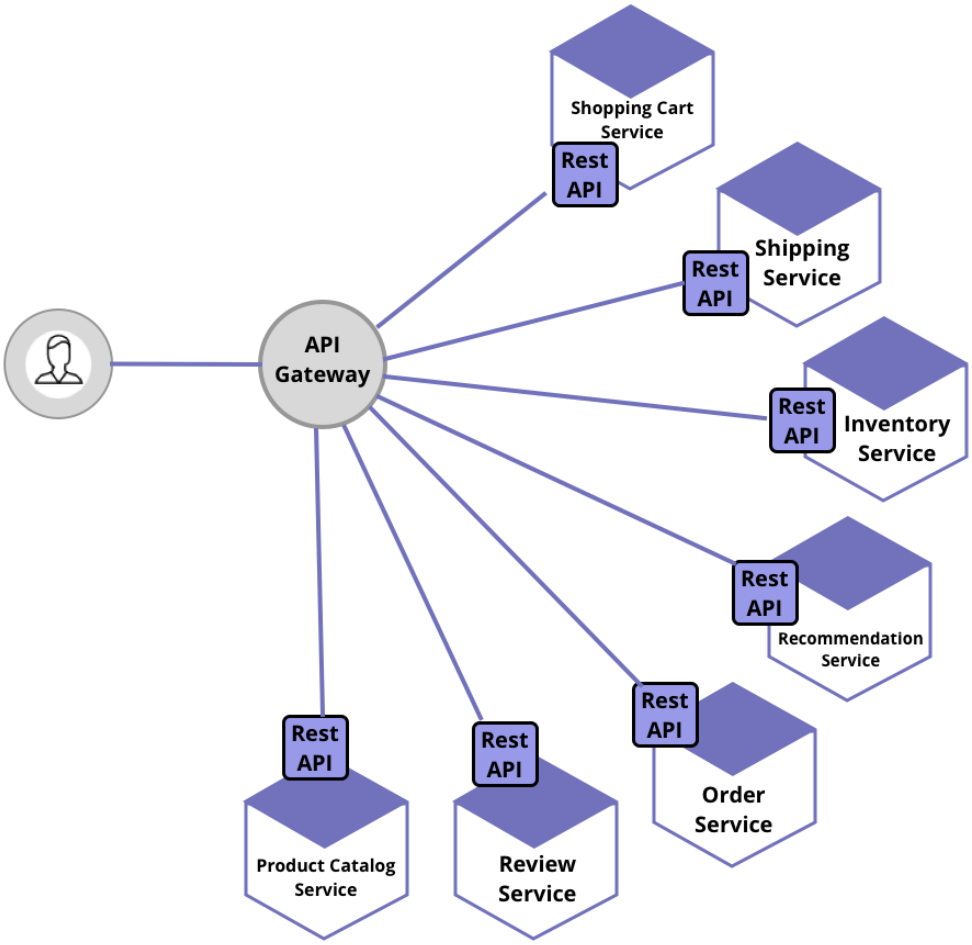

Figure 17: Service instance per container pattern (VMs) (Richardson & Smith, 2016).

Netflix in the current market uses this API gateway technology to integrate their services. It provides service to various types of clients, such as mobile clients, TV clients, web clients, and set-up box clients. In early stages, they designed a one-size-fits-all API which processes the request from every type of user. It does not satisfy their application very well due to different types of users and their unique needs. To overcome these challenges, they implemented API gateway in their architecture, which provides API, exposes the client requests, and handles them by invoking a device-based adapter code. The adaptor code service handles them by calling the required microservices.

The major advantage of the API gateway is that it simplifies the code. Rather than client requests invoking multiple services, it directly communicates with the API gateway which collects the responses and provides API to the client. With this, the client needs to send only a request to the application and the rest will be handled by a gateway, which reduces the number of request responses between them. Along with this advantage, there are also disadvantages to this approach. This is a highly dependent component which needs to be available all the time. Developing, deploying, and managing the gateway is another task needed to be handled by developers. Updating API is necessary to generate new and customized API for every microservice endpoint. Updating API gateway is an important task that should be handled with lightweight processes so that developers can keep their focus on application and gateway development. Every technology has advantages, drawbacks, and needs corrections and updates to overcome the difficulties, but using API gateway for the integration of microservices is presently one of the best practices available (Fowler, 2014).

Conclusion

This paper has discussed differences between the regular monolithic approach and service-oriented architecture, different types of service-based architecture, deployment and development issues, integration of microservices, and when they should be implemented. Some advantages and drawbacks have also been identified. Microservices makes our coding life easy by splitting up an application and helping to identify issues. The application can be scaled, monitored, and updated using small teams, which improves the quality of the application with continuous focus. This kind of architecture is booming in the IT world as it satisfies the many real-world applications which are in use now. It has provided many solutions to the issues faced

using monolithic and other SOAs, but it also has some backlogs in providing some core functionalities. Despite many positive reviews on MSA, it cannot be said that MSA is the right fit for every type of application. Though it has many advantages over the monolithic pattern, it requires more research to make a full judgment to say how prevalent microservices will be in the future.

Concern	Microservices	SOA
Deploy	Individual service deploy	Monolithic deploy, all at once
Teams	Microservices managed by individual teams	Services, Integration and user interface managed by individual teams
User Interface	Part of Microservices	Portal for all the services
Architecture Scope	One project	The whole company/enterprise
Flexibility	Fast independent service deploy	Business process adjustments on top services
Integration Mechanism	Simple and primitive integration	Smart and complex integration mechanism
Integration Technology	Heterogeneous if any	Homogenous, Single vendor
Cloud-native	Yes	No
Management/Governance Data Storage	Distributed Per Unit	Centralized Shared
Scalability	Horizontally better scalable. Elastic	Limited compared to µServices. Limited elasticity
Unit	Independently scalable	Shared database. Loosely coupled
Mainstream Communication	Choreography	Orchestration
Fit	Medium-sized infrastructure	Large Infrastructure
Service Size	Fine grained, small	Fine or coarse-grained
Versioning	More to open changes	Maintaining multiple services of different versions
Administration Level	Anarchy	Centralized
Business Rules Location	Particular services	Integration Component

Figure 18: Comparing microservices with SOA (Richards , 2016)

The real consequences of architecture can be seen only after a few years of development have been completed. It requires many more applications to be built with this pattern and run for possibly years to completely identify the nature of microservices. There are many reasons to say MSA is a poor design for some types of application, where boundaries of the components are unidentified. Shifting to MSA will not be an effective choice if the components inside the system are not compiled closely. Team skill is the most important factor in developing the application, regardless of how much stronger the architecture is implemented. A less skilled team will always produce poor systems and cannot say microservices can resolve the problem. One reasonable argument obtained from this research is that it is better to start an application using monolithic designs in which developers are familiar with; if this pattern exhibits problems later, based on the identification of the problem and the application nature, developers will have a clearer picture of how to shift this architecture to microservice-based to become more efficient. It is always hard to judge a technology in its early stages and it is difficult to come to a conclusion about one perfect architecture, as the software field is a continuously evolving industry.

References

Arsanjani, A., Ghosh, A., Allam, A., Abdollah, T., Ganapathy, S., & Holley, K. (2008). Soma: A method for developing service-oriented solutions. *IBM Systems Journal, 47*(3), 377-396.

Baklouti, F., Le Sommer, N., & Maheo, Y. (2017). *Choreography-based vs orchestration-based service composition in opportunistic networks*, pp. 1-8. doi:10.1109/WiMOB. 2017.8115771.

Bonilla, O. (2015). *The advantages and disadvantages of multiple, and hybrid repositories.* Los Gatos: BitKeeper.

Buyya, R. (2010). Cloud computing: The next revolution in information technology. *1st International Conference on Parallel Distributed and Grid Computing*, (pp. 2-3). Solan.

Danilo, A., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., … Sheridan, C. (2012). MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. *4th International Workshop on Modeling in Software Engineering (MISE)* (pp. 50-56).

Taibi, D., Lenarduzzi, V., & Pahl, C. D. (2018). *Architectural patterns for microservices: A systematic mapping study*. Finland: Tampere University of Technology.

Eeles, P. (2004). *The role of the software architect.* IBM Corporation.

Fowler, J. L. (2014, March). *Microservices*. Retrieved from martin fowler: http://martinfowler.com/articles/microservices.html

Franchitti, J.-C. (1992). *Software engineering G22.244-001*. Courant Institute of Mathematical Sciences.

Toffetti, G., Brunner, S., Blochlinger, M., Dudouet, F., & Edmonds, A. (2015). An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud* (pp. 19-24). ACM.

McGrath, G., & Brenner, P. R. (2017). Serverless computing: Design, implementation, and performance. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCSW),* Atlanta: IEEE.

Gill, S. (2015). *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the Cloud* (p. 583). Systems and Computing Engineering Department, Universidad de Los Andes, Bogotá, Colombia.

Gupta, A. (2013, January). *bcaq*. Retrieved from bcaq.blogspot.com: http://bcaq.blogspot.com/2013/01/advantages-and-disadvantages-of-layered.html

Hammouda, I. (2004). *Introduction to software architecture.* Gothenburg: Chalmers.

Hutchinson, J. (2007). Evolving existing systems to service-oriented architectures: Perspective and challenges. *IEEE International Conference on Web Services,* (pp. 896-903).

McGovern, J. (2006). Enterprise service oriented architectures. *Springer*.

Jamshidi, J. C. (2016). Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science,* (pp. 137-146). Rome.

Kahadawa, S. N. (2006). *Event-driven architecture: Achieving architectural agility.* Nittambuwa, Sri Lanka.

Kong. (n.d.). *Pattern: Multiple service instances per host.* Retrieved from http://microservices.io: http://microservices.io/patterns/deployment/multiple-services-per-host.html

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice.* Boston, MA: Addison-Wesley.

Lindgren, M. (2008). *Importance of software architecture during release planning*. IEEE/IFIP, WICSA.

Papazoglou, M., Traverso, P., Dastdar, S., & Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, *40*(11), 38-45.

McGovern, J. (2004). *A practical guide to enterprise architecture.* Prentice Hall.

Mordinyi, R. (2010). Space-based architectures as abstraction layer for distributed business. *2010 International Conference on Complex, Intelligent and Software Intensive Systems.* Vienna.

Morlion, P. (2018). *Software architecture: The 5 patterns*. DZone.

Mohammadi, M., & Mukhtar, M. (2013). A review of SOA modeling approaches for enterprise information systems. *Procedia Technology, 11*, 794-800.

Newman, S. (2015). *Building Microservices.* O'Reilly Media.

Niemela., L. D. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering.*

Continuous Delivery. (2012). InJ. Rossberg & M. Olausson (Eds.), *Pro application lifecycle management with visual studio* (pp. 425-432). Apress.

Lago, P., Malavotta, I., Muccini, H., Pellicione, P., & Tang, A. (2015). The road ahead for architectural languages. *IEEE Software, 32*, 98-105.

Castro, P., Ishakian, V., Muthusamy V., & Siominski, A. (2017). Serverless programming. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS),* Atlanta: IEEE.

Kazman, R., Klein, M. H., & Clements, P. C. (2000). *ATAM: Method for architecture evaluation.* Software Engineering Institute. CMU/USEI-2000-TR-004

Richards, M. (2016). *Microservices vs. service-oriented architecture.* CA: O'Reilly Media, Inc.

Richardson, C. (2015, May 19). *Nginx*. Retrieved from Introduction to Microservices: https://www.nginx.com/blog/introduction-to-microservices/

Richardson, C. (2016, Feb 23). *Integration zone*. Retrieved from dzone: https://dzone.com/articles/deploying-microservices

Richardson, C., & Smith, F. (2016). *Microservices from design to deployment.* NGINX.

Ruhe., M. O. (2005). Supporting software release planning decisions for evolving systems. *IEEE/NASA Software Engineering Workshop* (pp. 14-26). IEEE Computer Society.

Shaw, D. G. (1994). *An introduction to software architecture.* New Jersey: World Scientific Publishing Company.

Lorido-Botran, T., Minguel-Alonso, J., & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in Cloud environments. *Journal of Grid Computing*, pp. 559-592.

Ticki. (1998). *Redox book.* Retrieved from https://doc.redox-os.org/books/

Wayner, P. (2008). *5 software architecture patterns: How to make the right choice*. Independent.

Jia, S., Ying, S., Cao, H., & Xie, D. (2007). A new architecture description language for a service-oriented architect. In *Sixth International Conference on Grid and Cooperative Computing (GCC 2007),* (pp. 96-103). IEEE.