**PRACTICAL – 9**

**AIM**: Prepare the detailed case study on Service-Oriented Architecture (SOA) of AWS services, Microsoft Azure services, Google Cloud Platform services as Cloud Computing Software Engineering developer in IT industry.

**THEORY**

Service-Oriented Architecture (SOA) is an architectural approach in which applications make use of services available in the network. In this architecture, services are provided to form applications, through a communication call over the internet.

* SOA allows users to combine a large number of facilities from existing services to form applications.
* SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system.
* SOA based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.

There are two major roles within Service-oriented Architecture:

* Service provider:
  + The service provider is the maintainer of the service and the organization that makes available one or more services for others to use. To advertise services, the provider can publish them in a registry, together with a service contract that specifies the nature of the service, how to use it, the requirements for the service, and the fees charged.
* Service consumer:
  + The service consumer can locate the service metadata in the registry and develop the required client components to bind and use the service.
  + Services might aggregate information and data retrieved from other services or create workflows of services to satisfy the request of a given service consumer. This practice is known as service orchestration Another important interaction pattern is service choreography, which is the coordinated interaction of services without a single point of control.

**Advantages of SOA:**

* + Service reusability - In SOA, applications are made from existing services. Thus, services can be reused to make many applications.
  + Easy maintenance - As services are independent of each other they can be updated and modified easily without affecting other services.
  + Platform independent - SOA allows making a complex application by combining services picked from different sources, independent of the platform.
  + Availability - SOA facilities are easily available to anyone on request.
  + Reliability - SOA applications are more reliable because it is easy to debug small services rather than huge codes
  + Scalability - Services can run on different servers within an environment, this increases scalability

**Disadvantages of SOA:**

* High overhead:
  + A validation of input parameters of services is done whenever services interact this decreases performance as it increases load and response time.
* High investment:
  + A huge initial investment is required for SOA.
* Complex service management:
  + When services interact, they exchange messages to tasks. the number of messages may go in millions. It becomes a cumbersome task to handle a large number of messages.

**Practical applications of SOA:**

SOA is used in many ways around us whether it is mentioned or not.

* SOA infrastructure is used by many armies and air force to deploy situational awareness systems.
* SOA is used to improve the healthcare delivery.
* Nowadays many apps are games and they use inbuilt functions to run. For example, an app might need GPS so it uses inbuilt GPS functions of the device. This is SOA in mobile solutions.
* SOA helps maintain museums a virtualized storage pool for their information and content.

**CASE STUDY ON SOFTWARE ORIENTED ARCHITECTURE(SOA) OF AWS**

**Introduction:**

Service-Oriented Architecture (SOA) may be a kind of software design where services are provided to the opposite components by application components, through a communication protocol over a network. Its principles are independent of vendors and other technologies.

Implementations of SOA vary in terms of granularity: from only a few services that cover large areas of functionality to several dozens or many small applications in what’s termed “microservice” architecture.

**AWS Lambda:**

AWS Lambda may be a service offered by the Amazon Web Services platform. AWS Lambda allows you to upload code that may be run on an on-demand container managed by Amazon. AWS Lambda will manage the provisioning and managing of servers to run the code, so all that’s needed from the user may be a packaged set of code to run and some configuration options to define the context during which the server runs. These managed applications are mentioned as Lambda functions.

**Modes of Operation:**

AWS Lambda has 2 modes of operations.

* Asynchronous/Event-driven
  + Lambda functions is run in response to an occasion in asynchronous mode. Any source of events, such as S3, SNS, etc. won’t block and Lambda functions can make the most of this in some ways, like establishing a processing pipeline for a few chains of events.
  + There are many sources of data, and counting on the source events are pushed to a Lambda function from the event source, or polled for events by AWS Lambda.
* Synchronous/Request->Response
  + For applications that need a response to be returned synchronously, Lambda are often run-in synchronous mode.
  + Typically, this is often utilized in conjunction with a service called API Gateway to return HTTP responses from AWS Lambda to an end-user, however Lambda functions may be called synchronously via a right away call to AWS Lambda.

AWS Lambda functions are uploaded as a zipper file containing handler code additionally to any dependencies required for the operation of the handler.

**Lambda Functions as an Evolution of SOA:**

Basic SOA could be a thanks to structure your code-base into small applications so as to learn an application within the ways described earlier during this article. Arising from this, the tactic of communication between these applications comes into focus. Event-driven SOA (aka SOA 2.0) allows for not only the normal direct service-to-service communication of SOA 1.0, but also for events to be propagated throughout the architecture so as to speak change.

Event-driven architecture may be a pattern that naturally promotes loose coupling and composability. By creating and reacting to events, services are often added ad-hoc to feature new functionality to an existing event, and several other events is composed to supply richer functionality.

**CASE STUDY ON SOFTWARE ORIENTED ARCHITECTURE(SOA) OF MICROSOFT AZURE**

**Introduction:**

Azure is a new cloud computing platform under development by Microsoft (microsoft.com/windowsazure). Cloud computing allows developers to host applications in an Internet-accessible virtual environment. The environment transparently provides the hardware, software, network and storage needed by the application.

As with other cloud environments, Azure provides a hosted environment for applications. The added benefit of Azure is that .NET Framework applications can be deployed with minimal changes from their desktop siblings.

Applying service-oriented architecture (SOA) patterns and utilizing the experiences collected when implementing service-oriented solutions will be key to success when moving your services and applications into the new arena of cloud computing. To better understand how SOA patterns can be applied to Azure deployments, let’s take a look at a scenario in which a fictional bank moves its services to the cloud.

**Performance and Flexibility:**

After some stress testing, the Woodgrove Bank development team found that having only one central data store in SQL Azure led to slower and slower response times when traffic increased. The developers decided to address this performance issue by using Azure table storage, which is designed to improve scalability by distributing the partitions across many storage nodes. Azure table storage also provides fast data access because the system monitors usage of the partitions and automatically load-balances them. However, because Azure table storage isn’t a relational data store, the team had to design some new data storage structures and pick a combination of partition and row keys that would provide good response times.

**Messaging and Queuing:**

The objective is that no message should be lost even if services are offline due to error conditions or planned maintenance. The Asynchronous Queuing pattern allows this, though some offerings are not suitable for this pattern. For example, prompt answers with confirmation or denial of money transfers are necessary when dealing with online card transactions. But in another situation the pattern would do fine.

Communication between the Web and Worker roles is done with Azure Queues (as of the November CTP version it is possible to communicate directly between role instances), which are by default both asynchronous and reliable.

**Putting Queues to Work:**

As soon as a customer sends a message to UserAccountService, this message is placed in a Azure Queue and the customer receives a confirmation message. UserAccountWorker will then be able to get the message from the queue. Should UserAccountWorker be down, the message will not be lost as it is stored securely in the queue.

If the processing inside UserAccountWorker goes wrong, the message will not be removed from the queue. To ensure this, the call to the DeleteMessage method of the queue is made only after the work has been completed. If UserAccountWorker didn’t finish processing the message before the timeout elapsed (the timeout is hardcoded to 20 seconds), the message will again be made visible on the queue so that another instance of UserAccountWorker can attempt to process it.

As soon as a customer sends a message to UserAccountService, this message is placed in a queue and the customer receives a confirmation message of type TransactionResponse. From the perspective of the customer, Asynchronous Queuing is used. ReliableMessaging is used to communicate between UserAccountStorageAction and AccountStorageWorker, which reside in the Web role and Worker role, respectively.

**Processing Messages:**

The ProcessMessage method first needs to get the content of the message. This can be done in one of two ways.

First, the message could be stored as a string in the queue.

Second, the message could be serialized XML.

**Idempotent Capability:**

What if one of Woodgrove Bank’s customers sends a request to transfer money from one account to another and the message gets lost? If the customer resends the message, it is possible that two or more of the requests reach the services and gets treated separately.

One of the Woodgrove Bank team members immediately identified this scenario as one that requires the Idempotent Capability pattern. This pattern demands that capabilities or operations are implemented in such a way that they are safe to repeat. In short, the solution that Woodgrove Bank wants to implement requires well-behaved clients that attach a unique ID to each request and promise that they will resend the exact same message including the same unique ID in case of a retry. To be able to handle this, the unique ID is saved in the Azure table storage. Before processing any requests, it is necessary to check if a message with that ID was already processed. If it has been processed, a correct reply will be created, but the processing associated with the new request will not take place.

Although this means bothering the central data store with extra queries, it was deemed necessary. It will result in some deterioration of performance since some queries are made to the central data store before any other processing can take place. However, allowing this to consume extra time and other resources is a reasonable choice in order to meet Woodgrove Bank’s requirements.

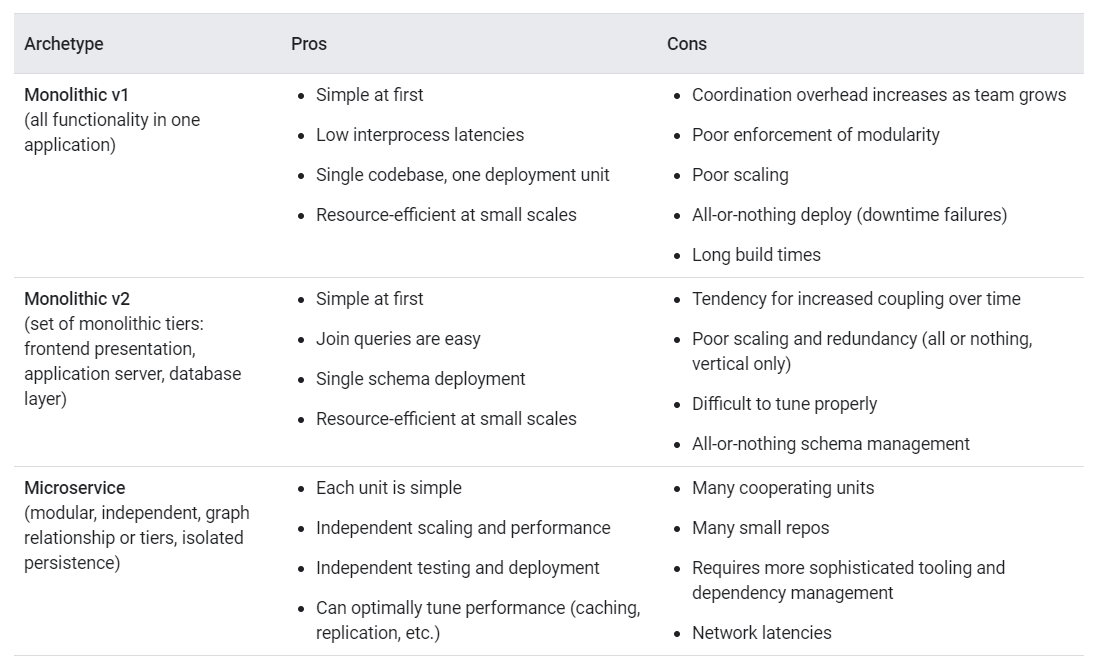
**CASE STUDY ON SOFTWARE ORIENTED ARCHITECTURE(SOA) OF GOOGLE CLOUD**

**Introduction:**

The principle behind service-oriented architecture (SOA) is to allow using applications as a service by other applications regardless of the platform (type of vendor, product, or technology) being used on either side (service and service consumer). This allows communication and data exchange between applications built on heterogeneous platforms without the need of any additional programming or making changes to the services. What we look for here is seamless integration between any type of applications.

Consider the major architectural archetypes. Randy Shoup, formerly an Engineering Director for App Engine and VP of Engineering at WeWork, observed that “There is no one perfect architecture for all products and all scales. Any architecture meets a particular set of goals or range of requirements and constraints, such as time to market, ease of developing functionality, scaling, etc. The functionality of any product or service will almost certainly evolve over time—it should not be surprising that our architectural needs will change as well. What works at scale 1x rarely works at scale 10x or 100x.”

Given the pros and cons of architectural archetypes, each fits a different evolutionary need for an organization.



As the table shows, a monolithic architecture that supports a lean product development effort (for example, rapid prototyping of new features, and potential pivots or large changes in strategies) is different from an architecture that needs hundreds of teams of developers, each of whom must be able to independently deliver value to the customer. By allowing the architecture to evolve, you can ensure that your architecture always serves the current needs of the organization. Regardless of the archetype, when architecting to facilitate continuous delivery, teams must be empowered to achieve the capabilities discussed in the introduction to this document.

Building cross-functional teams, with representation from across the organization (product, dev, test, and operations) enables teams to work independently and facilitates building around team boundaries. When your teams are cross-functional, they can function autonomously, experiment with ideas, and choose their own tools. To help with cross-team communication and testing, it can be helpful to have well-defined contracts between services.

Team independence is important, as is the independence of their products and services. Services need to be testable on demand. Adopting techniques around mocking and stubbing of external services helps reduce the impact of external dependencies and lets teams quickly create test environments. Also, implementing contract testing of external services helps ensure that dependencies on their service or other services are still met. To truly achieve continuous delivery, an individual team's product or service must be independently acceptance tested and deployed from the services it depends on.

To enable deploy-anytime capabilities, consider implementing blue/green or rolling deployment models, with high degrees of automation. With these models, at least two or more versions of the product or service are running simultaneously. These deployment models allow teams to validate changes and deploy to production with little or no downtime. An important consideration is how data upgrades are performed, meaning data and schema must be done in a backward-compatible manner.

In order to aid the independent deployment of components, we recommend that you create backward-compatible versioned APIs. Ensuring backward compatibility for APIs adds complexity to systems, but the flexibility you gain in terms of ease of deployment pays for the added complexity many times over.

Service-oriented and microservice architectures enable these capabilities because they use bounded contexts and APIs as a way to decouple large domains into smaller, more loosely coupled units and the use of test doubles and virtualization as a way to test services or components in isolation.

**Common pitfalls in architectures:**

* Simultaneously releasing many services.
  + In teams where testability and deployability are not prioritized, most testing requires the use of complex and expensive integrated environments. In many cases, deployments require that you simultaneously release many services due to complex interdependencies. These "big-bang" deployments require teams to orchestrate their work, with many hand-offs and dependencies between hundreds or thousands of tasks. Big-bang deployments typically take many hours or even days, and require scheduling significant downtime.
* Integrating changes with the changes from hundreds, or even thousands, of other developers.
  + Those developers, in turn, might have dependencies on tens, hundreds, or thousands of interconnected systems. Testing is done in scarce integration test environments, which often require weeks to obtain and configure. These environments are typically not representative of production, reducing the value and accuracy of the testing. The result is not only long lead times for changes (typically measured in weeks or months) but also low developer productivity and poor deployment outcomes.
* Creating bottlenecks in the software delivery process.
  + Example bottlenecks could be a single team that many others rely on either from a manual process standpoint (testing, deployment, and so on) or from a service operation standpoint. In both examples, those bottlenecks create single points of failure and demand that those teams or services scale to meet the demands of the many dependent teams.

**Advantages:**

Metrics and monitoring become more important and escalations become more difficult because an issue surfaced in one service could be from a service many service calls away.

Internal services can produce Denial of Service (DOS) type problems, so quotas and message throttling are important in every service.

QA and monitoring begin to blend, because monitoring must be comprehensive and must exercise the business logic and data of the service.

When there are many services, having a service-discovery mechanism becomes important for efficient operation of the system.

Without a universal standard for running a service in a debuggable environment, debugging issues in other people's services is much harder.

**CONCLUSION:**

In this practical, we learned about service-oriented architecture (SOA). We studied about its advantages and disadvantages. We learned about different platforms like AWS offered by Amazon, Azure offered by Microsoft and Google cloud offered by Google.