**Question-4:**

The compute stack offers functionality that is implemented across a set of microservices. Top-level functionality, such as “create a VM” is implemented by calling into several of these microservices. As described in the introduction to this application, Azure Stack appliances can be configured with 4 to 16 servers and our microservices are distributed across these servers. It can take several minutes to provision a VM.

What are some of the concerns to consider with this design? What are some strategies for mitigating those concerns?

**Answer:**

I have tried to explore and explain concerns and what strategies can be implemented to mitigate those concerns.

1. **Concern-1: Performance issues**
2. **Latency**–

As we have a distributed microservice architecture and the creating VM service has been distributed over 4-16 servers, the entire service call completes by communicating with the several microservices which leads to the latency.

* Network Latency-
* The servers may be deployed into the same region or different region hence the communication between the microservices happens over the network which adds the delay into the overall response time of operation.
* Service Discovery Latency**-**
* A single microservice can be accessed by several application at a time and all the applications need to discover that service. To achieve service discovery needs a broker which links microservices with the applications. The API brokerage process is based on a middleware tool also add delay.

1. **Controlling latency for scalability-**

* In our case, scaling microservices across the range of servers creates the scalability challenge.
* Scaling microservice introduce a new instance of the microservices in different regions which add the unpredictable latency.
* It also needs load balancing and load balancer’s in one region and microservices into a different location leads to the unpredictable latency.

**Mitigation Strategies- Performance issues**

In our case of creating VM which takes several minutes to provision a VM, latency between the microservices communication is a key reason for the taking several minutes to provision VM.

I came up with below mitigation strategies to overcome the performance hit caused by latency.

1. **Examine workflow-**

* We can examine the situation where two or more microservices are highly dependent on each other and occur together in the workflow. In this case, we should combine the microservices and should avoid over componentizing.

1. **Examining the server’s hosting location and microservices communication-**

* We must examine the technical flow of our microservices communication and what are the server’s regions and how far they are hosted from each other, to avoid the network latency and improve the response time.
* For example- provision microservice operation is performed most of the time form the West coast region but the main microservice which provisions the VM is located into East coast region then it will add unnecessary latency.

1. **Examining shared components-**

* Examining shared resources or data centers are located geographically to the central location and connected with the faster network connection.
* Another approach we can use is, maintain multiple copies of the shared data into each region if that can be treated as a local copy of data. It can reduce latency and dependency on the data located in different regions.

1. **Can improve network connections between servers-**

* Latency can be minimized by improving the connections between servers which runs the microservices.
* Within the data centers, traditional switching can be improved with the fabric model which is non-blocking. I can be achieved without hampering the scalability goal.

1. **Load balancing and API brokerage-**

* API brokerage and load balancing can control the latency where we need to make sure the microservices are stateless whenever possible. Retrieving state information can cause a delay.
* If state information is necessary, then we must pay particular attention to where we store the state information. The state repository must be as local as possible.

1. **Concern-2: Remote procedure call (RPC) between Microservices**

* If we have synchronous communication between the microservices then we are adding dependencies in request and response.
* Synchronous calls block the resources and that add the delay in the overall operation of provisioning the VM.

**Mitigation Strategies- Asynchronous RPC Calls**

* To improve the performance of our VM provisioning service we must implement asynchronous RPC between the microservices.
* Asynchronous RPC doesn’t block the resources while we are waiting for the response and we can execute multiple services in parallel. This will improve the huge performance as we just have to wait until the slowest service to respond.

**Sample Asynchronous REST RPC calls using JAX-RS**

@Stateless

@Path("/virtualMachine")

public class BookEndpoint {

@Resource

ManagedExecutorService exec;

@GET

@Path("/provisionVM")

public void provisionVM (@Suspended AsyncResponse response) {

response.setTimeout(5, TimeUnit.SECONDS);

log.info("Thread.currentThread().getName()”);

exec.execute(new Runnable() {

@Override

public void run() {

log.info("Thread.currentThread().getName()”);

// Business logic to provision Virtual Machine

// resume request and return result

response.resume(Response.ok("Provisioned VM").build());

}

});

}

}

1. **Concern-3: Resiliency and Fault Tolerance**

* Another concern about our distributed microservices can majorly affect the functioning and provisioning of VM is the failure of one of the microservice.
* As there is a number of moving components in a distributed microservices architecture, it has more point of failure as well.
* Failure can be caused by many reasons including errors and exceptions in the code, the release of new code, defects in the new released code, hardware failure, lack of unit testing and failed to handle the corner cases in the unit testing, dependent components broke due to modifying another code.

**Mitigation Strategies- Fault tolerance**

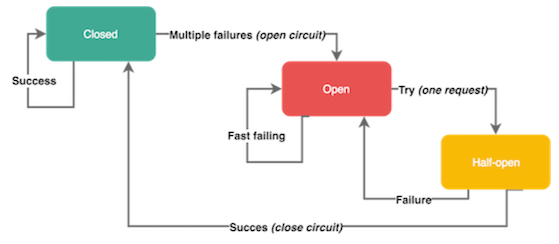
**How to make the microservices resilient?**

1. **Design for failure**

* Implementing microservices architecture by considering some design patterns can ensure resiliency.

1. **Circuit Breaker Pattern**

* This design pattern helps to fail fast by opening the circuit when there is a failure in microservices.
* It ensures that no additional calls are made to the failed service once the circuit breaker is opened. Means it returns an exception immediately once the call made to the failed service.
* This pattern also implemented to monitor the system for failures and once the things are back to normal, the circuit will be closed to allow normal functioning.



1. **Retry Design Pattern**

* If one of the microservice which provision VM get failed due to the network issue, then retry connection pattern can be a handy fix to the problem.
* Many times, the simple restart of the services can fix the issue. The load balancer may point the service call to the healthy server and call might be successful.

1. **Identify failure scenario**

* Before releasing our new microservice to the production or after modifying the existing microservice we must test it enough for the failure scenarios.
* Beforehand identifying the scenario’s where our microservice can fail and what are the ways we can recover it.

**For example:**

1. Application is not able to connect with the File System.
2. Microservices A failing to connect with microservice B
3. The database is not accessible.
4. Intentionally add some delays into the services and observe the performance issues.
5. **Avoid cascading failures**

* If our architecture has service dependency built inside, then we must ensure that the failure of one of our microservice does not affect other services in a network.

1. **Avoid single point of failures**

* During the development of each of the microservice, we must think and implement the solution for the question of **How the entire system will still work if this microservice fails?**

1. **Failure handling – Allow fast degradation**

* We must handle the errors/exception in such a way that it gracefully handles the failure by providing error messages or default values to proceed with.