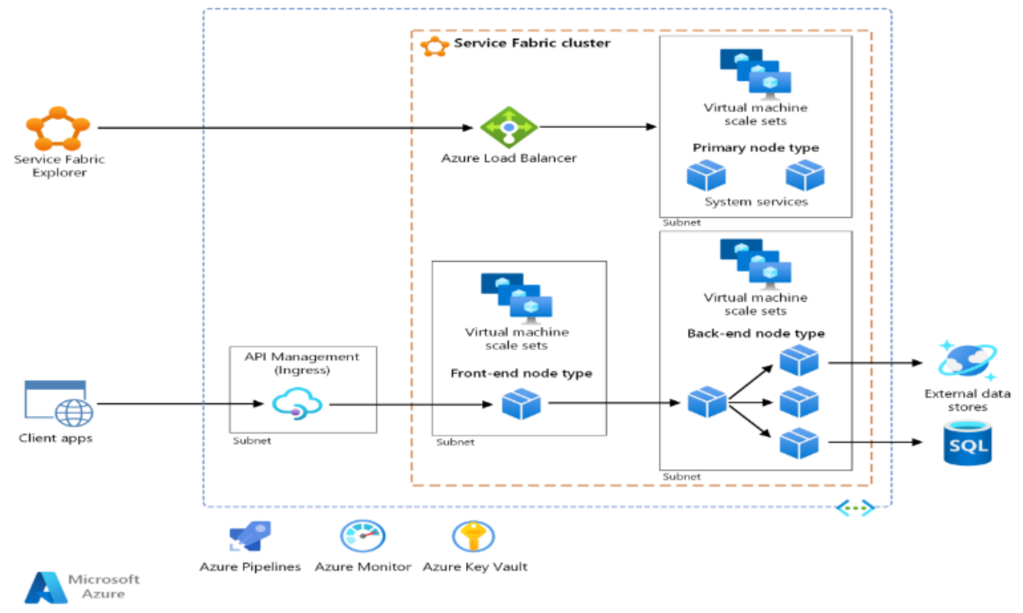
Microservices with -Azure Service Fabric

Problem Statement:

As applications grow in complexity and scale, they often need to be decomposed into smaller, independently deployable units called microservices. This can help improve the maintainability, scalability, and resilience of the overall system. However, building and managing a large number of microservices can be challenging, especially when it comes to ensuring that they are scalable and resilient. Microservices need to be able to handle increased workloads, communicate with each other in a reliable and efficient manner, and recover from failures. In addition, managing a microservices architecture requires tools and infrastructure to deploy, monitor, and manage the individual services. These challenges can make it difficult to build and maintain a microservices-based system, and can hinder the benefits that microservices can provide.

Solution Architecture:



Azure Service Fabric serves as an optimal solution for constructing scalable and resilient microservices. Functioning as a microservices platform, it empowers users to construct and oversee scalable and dependable applications. Noteworthy features include a distributed systems runtime, which efficiently manages the underlying infrastructure, allowing developers to concentrate on application development. The platform also offers a simplified programming model that abstracts common microservices tasks like communication and state management. Complementing these features are a set of tools and APIs dedicated to the seamless deployment, scaling, and monitoring of microservices.

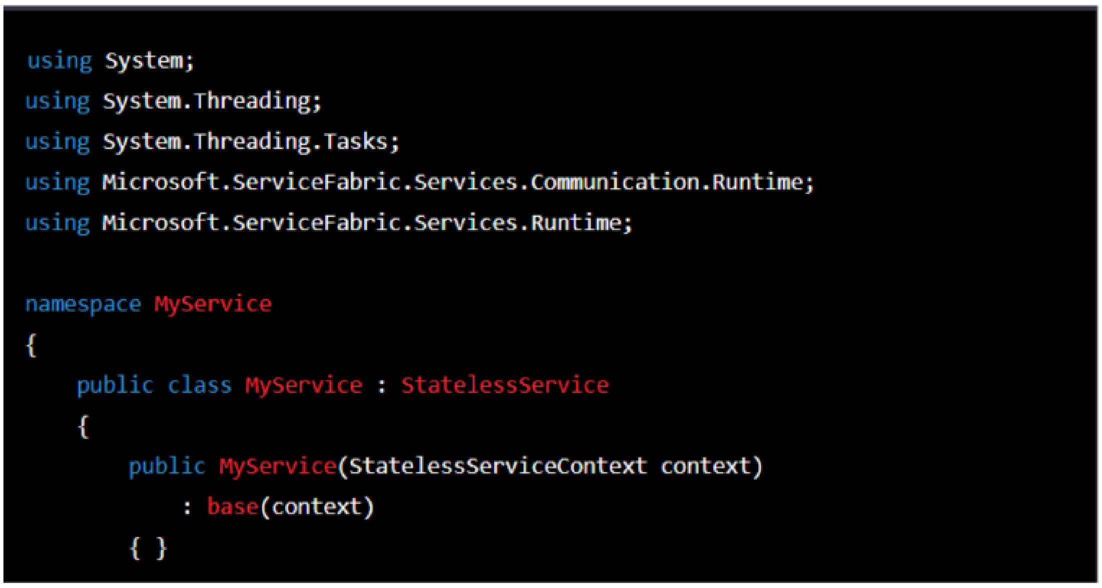
Technical Details and Implementation:

To build a microservice with Service Fabric, you first need to create a Service Fabric application. This is a logical container for your microservices that defines the overall structure and behavior of your application.

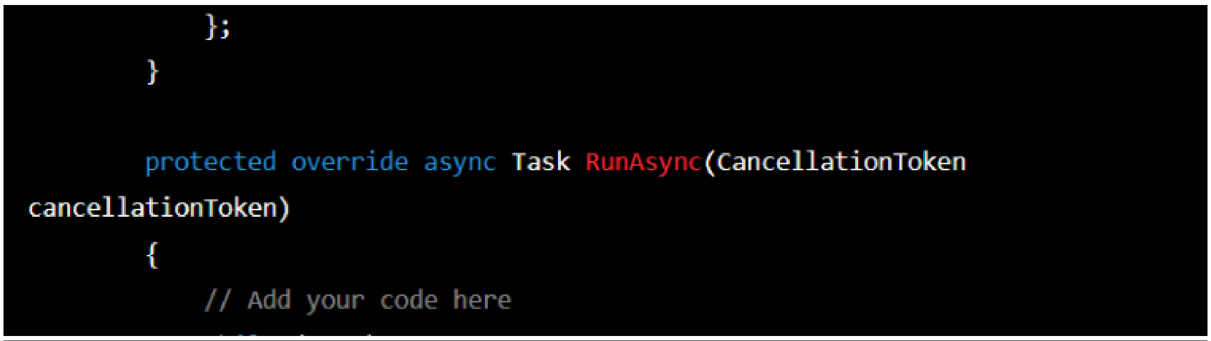
```
using Microsoft.ServiceFabric.Services.Runtime;

namespace MyService
{
    internal static class Program
    {
        private static void Main()
        {
            ServiceRuntime.RegisterServiceAsync("MyServiceType", context => new
MyService(context)).GetAwaiter().GetResult();

            Thread.Sleep(Timeout.Infinite);
        }
    }
}
```

Next, you can create a microservice by defining a stateless or stateful service. A stateless service is a microservice that does not maintain any state and can be scaled out horizontally to handle increased workload. A stateful service is a microservice that maintains a state and can be scaled out or in based on the workload.

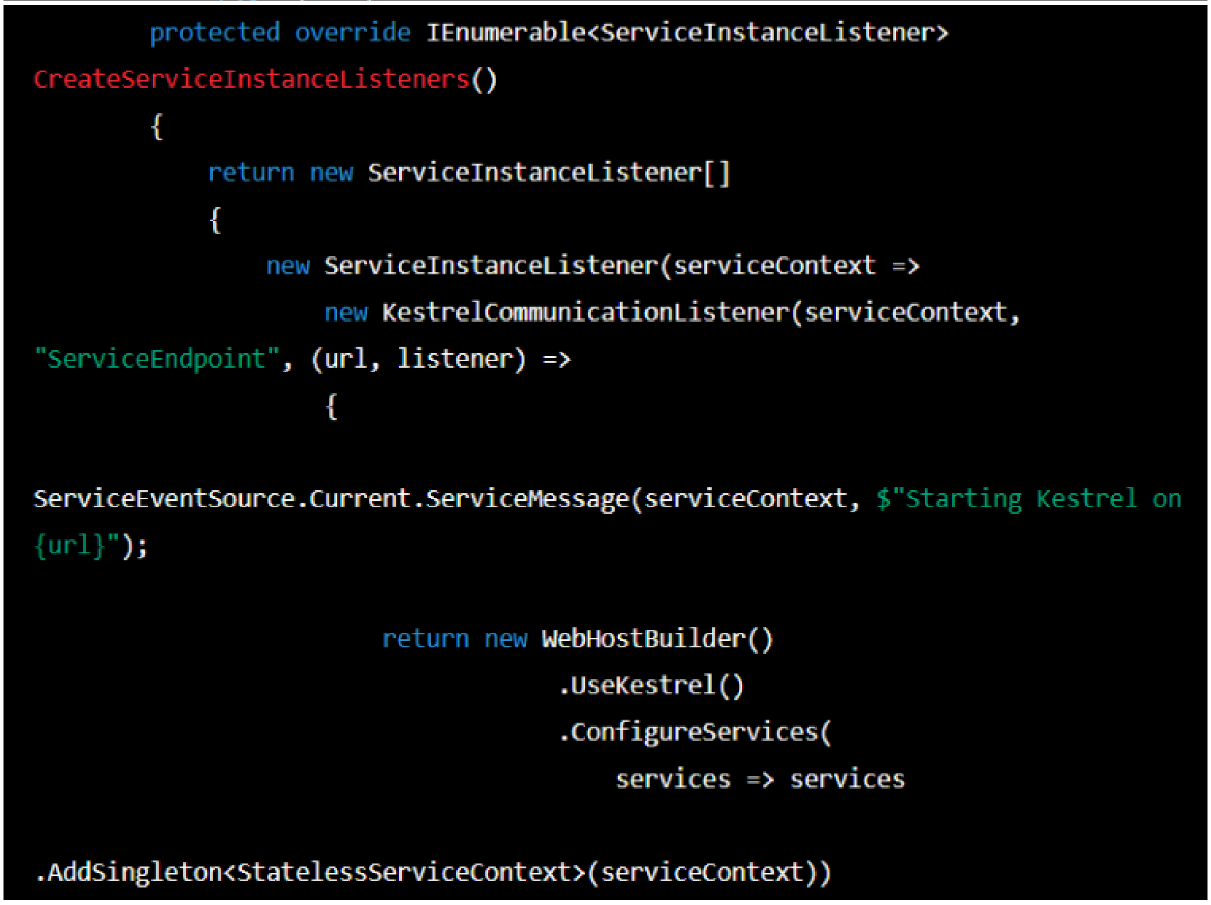
Here is an example of a stateless service in Azure Service Fabric using the .NET Core programming model:



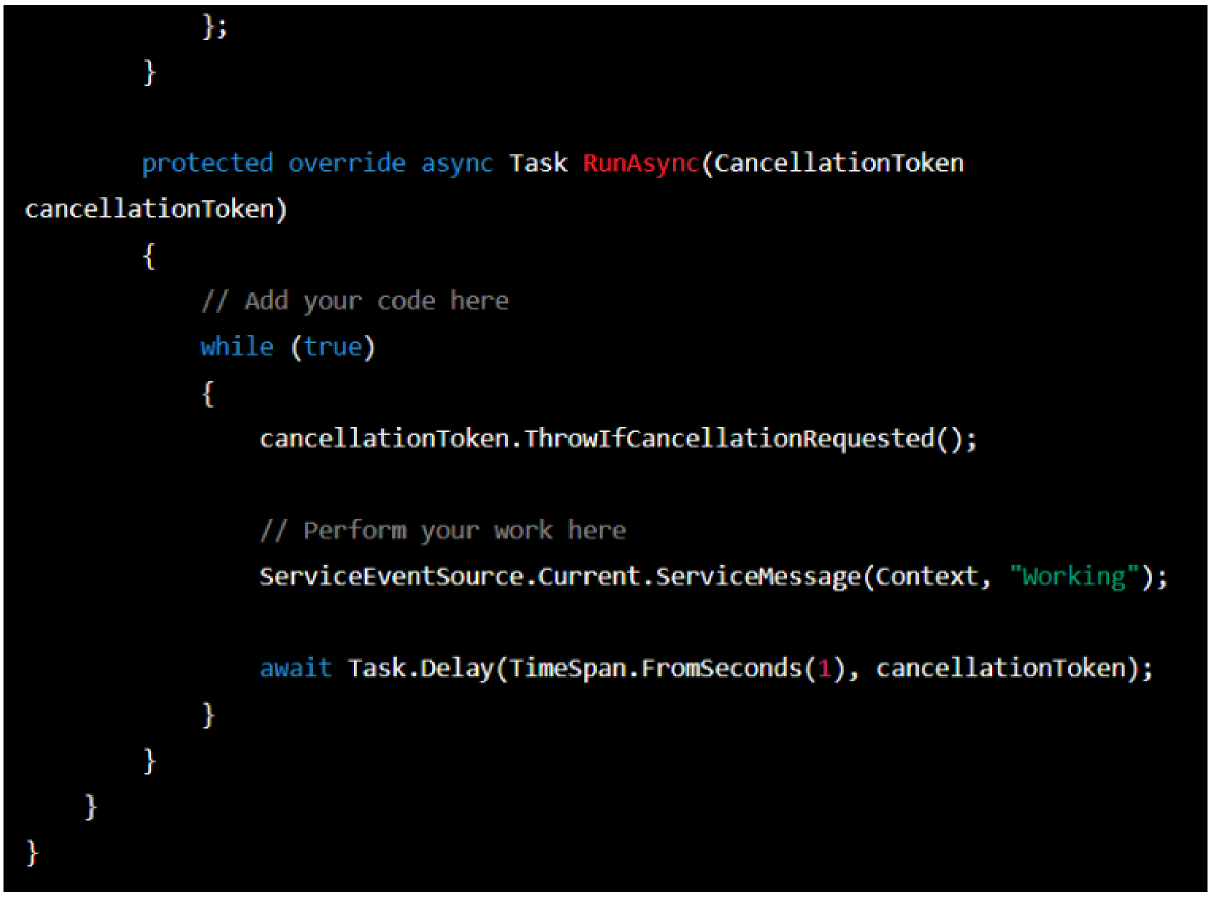
```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Microsoft.ServiceFabric.Services.Runtime;

namespace MyService
{
    public class MyService : StatelessService
    {
        public MyService(StatelessServiceContext context)
            : base(context)
        { }
```



```
            };
        }

        protected override async Task RunAsync(CancellationToken
cancellationToken)
        {
            // Add your code here
```



```
        protected override IEnumerable<ServiceInstanceListener>
CreateServiceInstanceListeners()
        {
            return new ServiceInstanceListener[]
            {
                new ServiceInstanceListener(serviceContext =>
                    new KestrelCommunicationListener(serviceContext,
"ServiceEndpoint", (url, listener) =>
                    {

ServiceEventSource.Current.ServiceMessage(serviceContext, $"Starting Kestrel on
{url}");

                        return new WebHostBuilder()
                                    .UseKestrel()
                                    .ConfigureServices(
                                        services => services

.AddSingleton<StatelessServiceContext>(serviceContext))
```

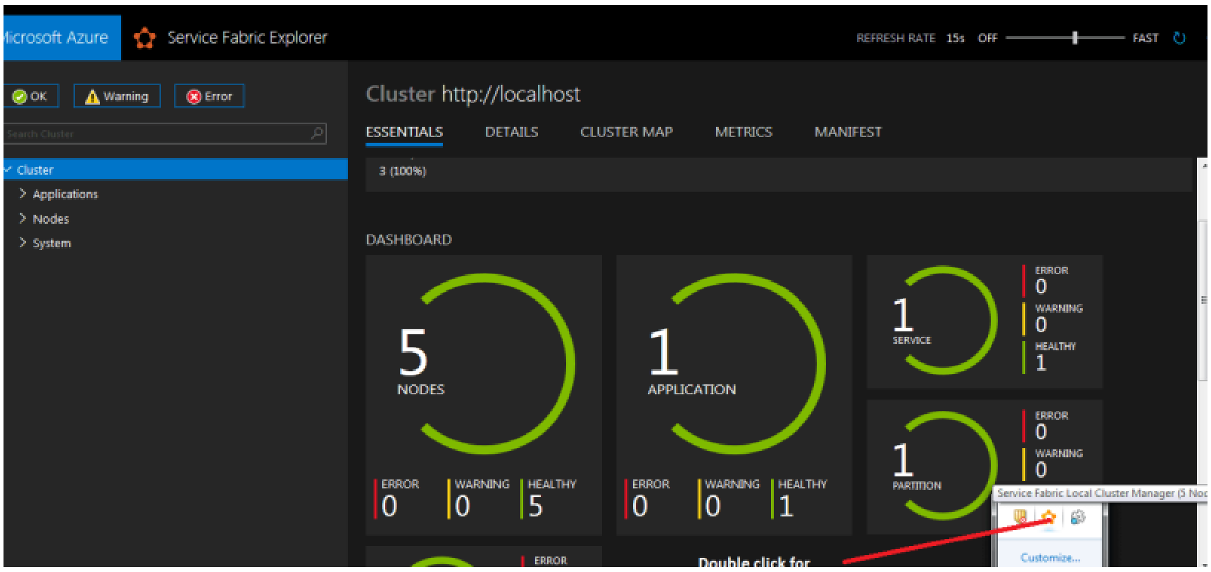


```
            };
        }

        protected override async Task RunAsync(CancellationToken
cancellationToken)
        {
            // Add your code here
            while (true)
            {
                cancellationToken.ThrowIfCancellationRequested();

                // Perform your work here
                ServiceEventSource.Current.ServiceMessage(Context, "Working");

                await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
            }
        }
    }
}
```

This code establishes a stateless service class named MyService, inheriting from StatelessService. The overridden CreateServiceInstanceListeners method is pivotal for generating a listener facilitating communication with external clients—here, utilizing a Kestrel listener.

Upon defining the microservice, the next step involves implementing logic through the Service Fabric programming model. This includes scripting code to handle incoming requests, interact with other microservices, and execute necessary tasks. In the provided example, the RunAsync method undertakes specific tasks, such as logging a message to the Service Fabric event log every second. Tailoring the service's behavior involves inserting custom code into the RunAsync method to meet specific requirements.

Service Fabric also provides a number of tools and APIs that you can use to deploy, scale, and monitor your microservices. For example, you can use the Service Fabric Explorer to view the status of your microservices and the underlying infrastructure



Challenges Faced:

Implementing a microservices architecture with Service Fabric presents challenges, notably in managing a multitude of independent services. To address this, leveraging tools like Service Fabric Explorer and Service Fabric REST APIs proves essential for effective monitoring and management of microservices.

Another challenge arises in the potential for increased latency during communication between microservices. However, this challenge can be mitigated by optimizing communication patterns and utilizing features such as the Service Fabric Reliable Services and Reliable Actors programming models.

For instance, the Reliable Services programming model offers abstractions for constructing stateless or stateful services with reliable messaging patterns, facilitating seamless communication. Similarly, the Reliable Actors programming model provides a set of abstractions tailored for building actor-based systems, optimizing interactions among actors with their own states using reliable messaging.

Business Benefits:

Leveraging Azure Service Fabric for building scalable and resilient microservices yields numerous business advantages. Enhanced scalability is a key benefit, enabling seamless expansion of microservices to accommodate growing workloads effortlessly. Additionally, Service Fabric contributes to improved reliability through features like automatic failover and self-healing, ensuring consistent availability of microservices.

Furthermore, adopting Service Fabric results in time and resource savings, given its capability to manage the underlying infrastructure autonomously. The platform provides a suite of tools and APIs dedicated to microservices management, allowing organizations to concentrate on application development and enhancement rather than being concerned with infrastructure intricacies.