


# Microservices architecture on Azure Service Fabric

06/13/2019 • 24 minutes to read • Contributors 

## In this article

[Architecture](#)

[Design considerations](#)

[Scalability considerations](#)

[Availability considerations](#)

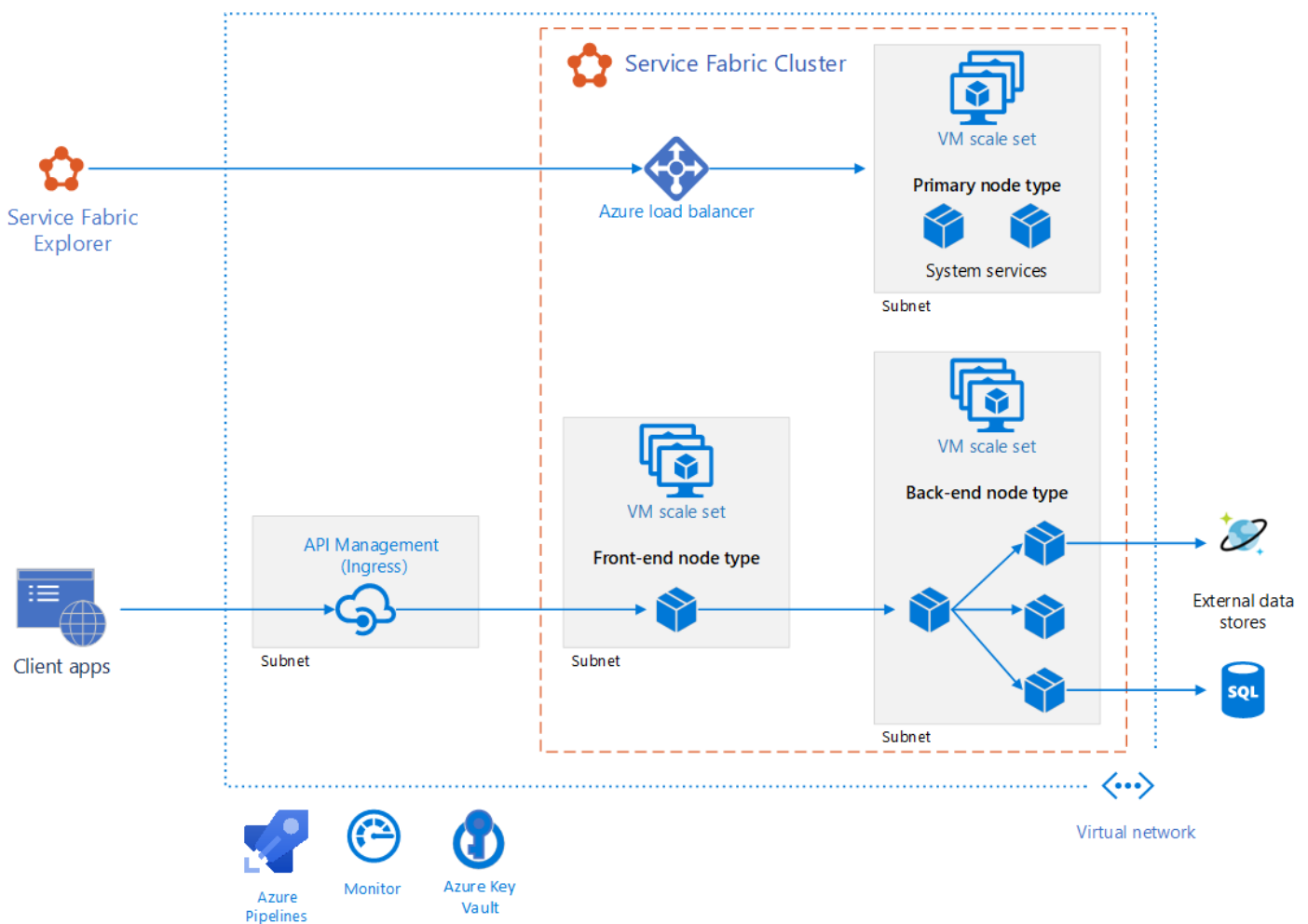
[Security considerations](#)

[Resiliency considerations](#)

[Monitoring considerations](#)

[Next steps](#)

This reference architecture shows a microservices architecture deployed to Azure Service Fabric. It shows a basic cluster configuration that can be the starting point for most deployments.



## ! Note

This article focuses on the [Reliable Services](#) programming model for Service Fabric. Using Service Fabric to deploy and manage [containers](#) is beyond the scope of this article.

# Architecture

The architecture consists of the following components. For other terms, see [Service Fabric terminology overview](#).

**Service Fabric cluster.** A network-connected set of virtual machines (VMs) into which your microservices are deployed and managed.

**Virtual machine scale sets.** Virtual machine scale sets allow you to create and manage a group of identical, load balanced, and autoscaling VMs. It also provides the fault and upgrade domains.

**Nodes.** The nodes are the VMs that belong to the Service Fabric cluster.

**Node types.** A node type represents a virtual machine scale set that deploys a collection of nodes. A Service Fabric cluster has at least one node type. In a cluster with multiple node types, one must be declared the [Primary node type](#). The primary node type in the cluster runs the [Service Fabric system services](#). These services provide the platform capabilities of Service Fabric. The primary node type also acts as the [seed nodes](#) for the cluster, which are the nodes that maintain the availability of the underlying cluster. Configure [additional node types](#) to run your services.

**Services.** A service performs a standalone function that can start and run independently of other services. Instances of services get deployed to nodes in the cluster. There are two varieties of service in Service Fabric:

- **Stateless service.** A stateless service does not maintain state within the service. If state persistence is required, then state is written to and retrieved from an external store, such as Azure Cosmos DB.
- **Stateful service.** The [service state](#) is kept within the service itself. Most stateful services implement this through Service Fabric's [Reliable Collections](#).

**Service Fabric Explorer.** [Service Fabric Explorer](#) is an open-source tool for inspecting and managing Service Fabric clusters.

**Azure Pipelines.** [Pipelines](#) is part of [Azure DevOps Services](#) and runs automated builds, tests, and deployments. You can also use third-party CI/CD solutions such as Jenkins.

**Azure Monitor.** [Azure Monitor](#) collects and stores metrics and logs, including platform metrics for the Azure services in the solution and application telemetry. Use this data to monitor the application, set up alerts and dashboards, and perform root cause analysis of failures. Azure Monitor integrates with Service Fabric to collect metrics from controllers, nodes, and containers, as well as container logs and master node logs.

**Azure Key Vault.** Use [Key Vault](#) to store any application secrets used by the microservices, such as connection strings.

**Azure API Management.** In this architecture, [API Management](#) acts as an API gateway that accepts requests from clients and routes them to your services.

## Design considerations

This reference architecture is focused on [microservices architectures](#). A microservice is a small, independently versioned unit of code. It is discoverable through service discovery mechanisms and can communicate with other services over APIs. Each service is self-contained and should implement a single business capability. For more information about how to decompose your application domain into microservices, see [Using domain analysis to model microservices](#).

Service Fabric provides an infrastructure to build, deploy, and upgrade microservices efficiently. It also provides options for auto scaling, managing state, monitoring health, and restarting services in case of failure.

Service Fabric follows an application model where an application is a collection of microservices. The application is described in an [application manifest](#) file that defines the different types of service contained in that application, and pointers to the independent service packages. The application package also usually contains parameters that serve as overrides for certain settings used by the services. Each service package has a manifest file that describes the physical

files and folders that are necessary to run that service, including binaries, configuration files, and read-only data for that service. Services and applications are independently versioned and upgradable.

Optionally, the application manifest can describe services that are automatically provisioned when an instance of the application is created. These are called default services. In this case, the application manifest also describes how these services should be created, including the service's name, instance count, security/isolation policy, and placement constraints.

#### ⓘ Note

Avoid using default services if you want to control the life time of your services. Default services are created when the application is created, and run as long as the application is running.

For more information about understanding Service Fabric, see [So you want to learn about Service Fabric?](#)

## Choose an application-to-service packaging model

A tenet of microservices is that each service can be independently deployed. In Service Fabric, if you group all of your services into a single application package, and one service fails to upgrade, the entire application upgrade gets rolled back, which prevents other service from being upgraded.

For that reason, in a microservices architecture, we recommend using multiple application packages. Put one or more closely related service types into a single application type. If your team is responsible for a set of services that run for the same duration and need to be updated at the same time, have the same lifecycle, or share resources such as dependencies or configuration, then place those services types in the same application type.

## Service Fabric programming models

When you add a microservice to a Service Fabric application, decide whether it has state or data that needs to be made highly available and reliable. If so, can it store data externally or is the data contained as part of the service? Choose a stateless service if you don't need to store data or want to store data in external storage. If you want to maintain state or data as part of the service (for example, you need that data to reside in memory close to the code), or cannot tolerate a dependency on an external store, consider choosing a stateful service.

If you have existing code that you want to run on Service Fabric, you can run it as a guest executable, which is an arbitrary executable that runs as a service. Alternatively, you can package the executable in a container that has all the dependencies needed for deployment. Service Fabric models both containers and guest executables as stateless services. For guidance about choosing a model, see [Service Fabric programming model overview](#).

With guest executables, you are responsible of maintaining the environment in which it runs. For example, suppose that a guest executable requires Python. If the executable is not self-contained, you need to make sure that the required version of Python is pre-installed in the environment. Service Fabric does not manage the environment. Azure offers multiple mechanisms to set up the environment, including custom virtual machine images and extensions.

To access a guest executable through a reverse proxy, make sure you have added the **UriScheme** attribute to the **Endpoint** element in the guest executable's service manifest.

 Copy

```
<Endpoints>
  <Endpoint Name="MyGuextExeTypeEndpoint" Port="8090" Protocol="http" UriScheme="http"
PathSuffix="api" Type="Input"/>
</Endpoints>
```

If the service has additional routes, specify the routes in the **PathSuffix** value. The value should not be prefixed or suffixed with '/'. Another way is to add the route in the service name.

 Copy

```
<Endpoints>
  <Endpoint Name="MyGuextExeTypeEndpoint" Port="8090" Protocol="http" PathSuffix="api"
Type="Input"/>
</Endpoints>
```

For more information, see:

- [Package an application](#)
- [Package and deploy an existing executable to Service Fabric](#)

## API gateway

An [API gateway](#) (ingress) sits between external clients and the microservices. It acts as a reverse proxy, routing requests from clients to microservices. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting.

Azure API Management is recommended for most scenarios, but [Træfik](#) is a popular open-source alternative. Both technology options are integrated with Service Fabric.

- API Management exposes a public IP address and routes traffic to your services. It runs in a dedicated subnet in the same virtual network as the Service Fabric cluster. It can access services in a node type that is exposed through a load balancer with a private IP address. This option is only available in the Premium and Developer tiers of API Management. For production workloads, use the Premium tier. Pricing information is described in [API Management pricing](#). For more information, see Service Fabric with [Azure API Management overview](#).
- Træfik supports features such as routing, tracing, logs, and metrics. Træfik runs as a stateless service in the Service Fabric cluster. Service versioning can be supported through routing. For information on how to set up Træfik for service ingress and as the reverse proxy within the cluster, see [Azure Service Fabric Provider](#). For more information about using Træfik with Service Fabric, see [Intelligent routing on Service Fabric with Træfik](#) (blog post).

Other API management options include [Azure Application Gateway](#) and [Azure Front Door](#). These services can be used in conjunction with API Management to perform tasks such as routing, SSL termination, and firewall.

## Interservice communication

To facilitate service-to-service communication, consider using HTTP as the communication protocol. As a baseline for most scenarios, we recommend using [the reverse proxy service](#) for service discovery.

- Communication protocol. In a microservices architecture, services need to communicate with each other with minimum coupling at runtime. To enable language-agnostic communication, HTTP is an industry-standard with a wide range of tools and HTTP servers that are available in different languages, all supported by Service Fabric. Therefore, using HTTP instead of Service Fabric's built in service remoting is recommended for most workloads.
- Service discovery. To communicate with other services within a cluster, a client service needs to resolve the target service's current location. In Service Fabric, services can move between nodes, causing the service endpoints to change dynamically. To avoid connections to stale endpoints, Service Fabric's Naming Service can be used to retrieve updated endpoint information. However, Service Fabric also provides a built-in [reverse proxy service](#) that abstracts the naming service. This option is easier to use and results in simpler code.

Other options for interservice communication include,

- [Træfik](#) for advanced routing.

- [DNS](#) for compatibility scenarios where a service expects to use DNS.
- [ServicePartitionClient<TCommunicationClient>](#) class. The class caches service endpoints and can enable better performance, as calls go directly between services without intermediaries or custom protocols.

## Scalability considerations

Service Fabric supports scaling these cluster entities:

- Scaling the number of nodes for each node type.
- Scaling services.

This section is focused on autoscaling. You can choose to manually scale in situations where appropriate. For example, a situation where manual intervention is required to set the number of instances.

### Initial cluster configuration for scalability

When you create a Service Fabric cluster, provision the node types based on your security and scalability needs. Each node type is mapped to a virtual machine scale set and can be scaled independently.

- Create a node type for each group of services that have different scalability or resource requirements. Start by provisioning a node type (which becomes the [primary node type](#)) for the Service Fabric system services. Then create separate node types to run your public or front-end services, and other node types as necessary for your backend and private or isolated services. Specify [placement constraints](#) so that the services are only deployed to the intended node types.
- Specify the durability tier for each node type. The durability tier represents the ability for Service Fabric to influence virtual machine scale set updates and maintenance operations. For production workloads, choose the Silver or higher durability tier. For information about each tier, see [The durability characteristics of the cluster](#).
- If using the Bronze durability tier, certain operations require manual steps. For node types with Bronze durability tier additional steps are required during scale in. For more information on scaling operations, see [this guide](#).

### Scaling nodes

Service Fabric supports autoscaling for scale-in and scale-out. Each node type can be configured for autoscaling independently.

Each node type can have a maximum of 100 nodes. Start with a smaller set of nodes and add more nodes depending on your load. If you require more than 100 nodes in a node type, you will need to add more node types. For details, see [Service Fabric cluster capacity planning considerations](#). A virtual machine scale set does not scale instantaneously, so consider that factor when you set up autoscale rules.

To support automatic scale-in, configure the node type to have the Silver or Gold durability tier. This makes sure that scaling in is delayed until Service Fabric is finished relocating services and that the virtual machine scale sets inform Service Fabric that the VMs are removed, not just down temporarily.

For more information about scaling at the node/cluster level, see [Scaling Azure Service Fabric clusters](#).

### Scaling services

Stateless and stateful services apply different approaches to scaling.

#### Autoscaling for stateless services

- Use the average partition load trigger. This trigger determines when the service is scaled in or out, based on a load threshold value specified in the scaling policy. You can also set how often the trigger is checked. See

[Average partition load trigger with instance-based scaling](#). This allows you to scale up to the number of available nodes.

- Set **InstanceCount** to -1 in the service manifest, which tells Service Fabric to run an instance of the service on every node. This approach enables the service to scale dynamically as the cluster scales. As the number of nodes in the cluster changes, Service Fabric automatically creates and deletes service instances to match.

#### ⓘ Note

In some cases, you might want to manually scale your service. For example, if you have a service that reads from Event Hubs, you might want a dedicated instance to read from each event hub partition, to avoid concurrent access to the partition.

## Scaling for stateful services

For a stateful service, scaling is controlled by the number of partitions, the size of each partition, and the number of partitions/replicas running on a given machine.

- If you are creating partitioned services, we recommend having a high number of partitions that are small in size. If more nodes are added, Service Fabric distributes the workloads onto the new machines by default. For example, if there are 5 nodes and 10 partitions, by default Service Fabric will place two primary replicas on each node. If you scale out the nodes, you can achieve greater performance, because the work is evenly distributed across more resources. For information about scenarios that take advantage of this strategy, see [Scaling in Service Fabric](#).
- Adding or removing partitions is not well supported. Another option commonly used to scale is to dynamically create or delete services or whole application instances. An example of that pattern is described in [Scaling by creating or removing new named services](#).

For more information, see:

- [Scale a Service Fabric cluster in or out using autoscale rules or manually](#)
- [Scale a Service Fabric cluster programmatically](#)
- [Scale a Service Fabric cluster out by adding a virtual machine scale set](#)

## Using metrics to balance load

Depending on how you design the partition, you might have nodes with replicas that get more traffic than others. To avoid this situation, partition the service state so that it is distributed across all partitions. Use the range partitioning scheme with a good hash algorithm. See [Get started with partitioning](#).

Service Fabric uses metrics to know how to place and balance services within a cluster. You can specify a default load for each metric associated with a service when that service is created. Service Fabric then takes that load into account when placing the service, or whenever the service needs to move (for example, during upgrades) to try to balance the nodes in the cluster.

The initially specified default load for a service will not change over the lifetime of the service. To capture changing metrics for a given service, we recommend that you monitor your service and then report the load dynamically. This allows Service Fabric to adjust the allocation based on the reported load at a given time. Use the [IServicePartition.ReportLoad](#) method to report custom metrics. For more information, see [Dynamic load](#).

## Availability considerations

Place your services in a node type other than the primary node type. The Service Fabric system services are always deployed to the primary node type. If your services are deployed to the primary node type, they might compete with system services for resources and interfere with the system services. If a node type is expected to host stateful services, make sure there are at least five node instances and you select the Silver or Gold Durability tier.

Consider constraining the resources of your services. See [Resource governance mechanism](#).

- Do not mix resource governed and resource non-governed services on the same node type. The non-governed services might consume too many resources, affecting the resource governed services. Specify [placement constraints](#) to make sure that those types of services do not run on the same set of nodes. See [Specify resource governance](#). (This is an example of the [Bulkhead pattern](#).)
- Specify the CPU cores and memory to reserve for a service instance. For information about usage and limitations of resource governance policies, see [Resource governance](#).

Make sure every service's target instance or replica count is greater than 1 to avoid a single point of failure (SPOF). The largest number that you can use as service instance or replica count equals the number nodes that to which the service is constrained.

Make sure every stateful service has at least two active secondary replicas. Five replicas are recommended for production workloads.

For more information, see [Availability of Service Fabric services](#).

## Security considerations

Here are some key points for securing your application on Service Fabric:

### Virtual network

Consider defining subnet boundaries for each virtual machine scale set to control the flow of communication. Each node type has its own virtual machine scale set in a subnet within the Service Fabric cluster's virtual network. Network Security Groups (NSGs) can be added to the subnets to allow or reject network traffic. For example, with front-end and back-end node types, you can add an NSG to the backend subnet to accept inbound traffic only the front-end subnet.

When calling external Azure Services from the cluster, use [Virtual Network service endpoints](#) if the Azure service supports it. Using a service endpoint secures the service to only the cluster's Virtual Network. For example, if you are using Cosmos DB to store data, configure the Cosmos DB account with a service endpoint to allow access only from a specific subnet. See [Access Azure Cosmos DB resources from virtual networks](#).

### Endpoints and interservice communication

Do not create an unsecured Service Fabric cluster. If the cluster exposes management endpoints to the public internet, anonymous users can connect to it. Unsecured clusters are not supported for production workloads. See: [Service Fabric cluster security scenarios](#).

To secure your interservice communications:

- Consider enabling HTTPS endpoints in your ASP.NET Core or Java web services.
- Establish a secure connection between the reverse proxy and services. For details, see [Connect to a secure service](#).

If you are using an [API gateway](#), you can [offload authentication](#) to the gateway. Make sure that the individual services cannot be reached directly (without the API gateway) unless additional security is in place to authenticate messages whether they come from the gateway.

Do not expose the Service Fabric reverse proxy publicly. Doing so causes all services that expose HTTP endpoints to be addressable from outside the cluster, introducing security vulnerabilities and potentially exposing additional information outside the cluster unnecessarily. If you want to access a service publicly, use an API gateway. Some options are mentioned in the [API gateway](#) section.

Remote desktop is useful for diagnostic and troubleshooting, but make sure not to leave it open otherwise it causes a security hole.

## Secrets and certificates

Store secrets such as connection strings to datastores in Azure Key Vault. The Key Vault must be in the same region as the virtual machine scale set. You will need to:

- Authenticate the service's access to the Key Vault.

Enable [managed identity](#) on the virtual machine scale set that hosts the service.

- Store your secrets in the Key Vault.

Add secrets in a format that can be translated to a key-value pair. For example, CosmosDB--AuthKey. When the configuration is built, "--" is converted into ":".

- Access those secrets in your service.

Add the Key Vault URI in your appSettings.json. In your service, add the configuration provider that reads from the Key Vault, builds the configuration, and accesses the secret from the built configuration.

Here's an example where the Workflow service stores a secret in the Key Vault in the format "CosmosDB--Database".

Copy

```
namespace Fabrikam.Workflow.Service
{
    public class ServiceStartup
    {
        public static void ConfigureServices(StatelessServiceContext context, IServiceCollection services)
        {
            var preConfig = new ConfigurationBuilder()
                .AddJsonFile(context, "appsettings.json");

            var config = preConfig.Build();

            if (config["AzureKeyVault:KeyVaultUri"] is var keyVaultUri && !string.IsNullOrWhiteSpace(keyVaultUri))
            {
                preConfig.AddAzureKeyVault(keyVaultUri);
                config = preConfig.Build();
            }
        }
    }
}
```

To access the secret, specify the secret name in the built config.

Copy

```
if(builtConfig["CosmosDB:Database"] is var database && !string.IsNullOrEmpty(database))
{
    // use the secret.
}
```

Do not use client certificates to access Service Fabric Explorer. Instead, use Azure Active Directory (Azure AD). Also see, [Azure services that support Azure AD authentication](#).

Do not use self-signed certificates for production.

## Data at rest protection



If you have attached data disks to the virtual machine scale sets of the Service Fabric cluster and your services save data on those disks, you must encrypt the disks. For more information, see [Encrypt OS and attached data disks in a virtual machine scale set with Azure PowerShell \(Preview\)](#).

For more information about securing Service Fabric, see:

- [Azure Service Fabric security overview](#)
- [Azure Service Fabric security best practices](#)
- [Azure Service Fabric security checklist](#)

## Resiliency considerations

To recover from failures and maintain a fully functioning state, the application must implement certain resiliency patterns. Here are some common patterns:

- [Retry pattern](#): To handle errors that are expected to be transient, such as resources being temporarily unavailable.
- [Circuit breaker](#): To address faults that might need longer to fix.
- [Bulkhead pattern](#): To isolate resources per service.

This reference implementation uses [Polly](#), an open source option, to implement all of those patterns.

## Monitoring considerations

Before you explore the monitoring options, we recommend you read this article about [diagnosing common scenarios with Service Fabric](#). You can think of monitoring data in these sets:

- [Application metrics and logs](#)
- [Service Fabric health and event data](#)
- [Infrastructure metrics and logs](#)
- [Metrics and logs for dependent services](#)

These are the two main options for analyzing that data:

- Application Insights
- Log Analytics

You can use Azure Monitor to set up dashboards for monitoring and to send alerts to operators. There are also some third-party monitoring tools that are integrated with Service Fabric, such as Dynatrace. For details, see [Azure Service Fabric Monitoring Partners](#).

### Application metrics and logs


Application telemetry provides data about your service that can help you monitor the health of your service and identify issues. To add traces and events in your service:

- [Microsoft.Extensions.Logging](#) if you are developing your service with ASP.NET Core. For other frameworks, use a logging library of your choice such as Serilog.
- You can add your own instrumentation by using the [TelemetryClient](#) class in the SDK and view the data in Application Insights. See [Add custom instrumentation to your application](#).
- Log ETW events by using [EventSource](#). This option is available by default in a Visual Studio Service Fabric solution.

Application Insights provides a lot of built-in telemetry: requests, traces, events, exceptions, metrics, dependencies. If your service exposes HTTP endpoints, enable Application Insights by calling the **UseApplicationInsights** extension method for [Microsoft.AspNetCore.Hosting.IWebHostBuilder](#). For information about instrumenting your service for Application Insights, see these articles:

- [Tutorial: Monitor and diagnose an ASP.NET Core application on Service Fabric using Application Insights.](#)
- [Application Insights for ASP.NET Core](#)
- [Application Insights .NET SDK](#)
- [Application Insights SDK for Service Fabric](#)

To view the traces and event logs, use [Application Insights](#) as one of sinks for structured logging. Configure Application Insights with your instrumentation key by calling the **AddApplicationInsights** extension method. In this example, the instrumentation key is stored as a secret in the Key Vault.

	 Copy
<pre>.ConfigureLogging((hostingContext, logging) =&gt; {     logging.AddApplicationInsights(hostingContext.Configuration ["ApplicationInsights:InstrumentationKey"]); })</pre>	

If your service does not expose HTTP endpoints, you need to write a custom extension that sends traces to Application Insights. For an example, see the Workflow service in the reference implementation.

ASP.NET Core services use the [ILogger interface](#) for application logging. To make these application logs available in Azure Monitor, send the ILogger events to Application Insights. For more information, see [ILogger in an ASP.NET Core application](#). Application Insights can add correlation properties to ILogger events, which is useful for visualizing distributed tracing.

For more information, see:

- [Application logging](#)
- [Add logging to your Service Fabric application](#)

## Service Fabric health and event data

Service Fabric telemetry includes health metrics and events about the operation and performance of a Service Fabric cluster and its entities: its nodes, applications, services, partitions, and replicas.

- **EventStore.** A stateful system service that collects events related to the cluster and its entities. Service Fabric uses EventStore to write [Service Fabric events](#) to provide information about your cluster and can be used for status updates, troubleshooting, monitoring. It can also correlate events from different entities at a given time to identify issues in the cluster. The service exposes those events through a REST API. For information about how to query the EventStore APIs, see [Query EventStore APIs for cluster events](#). You can view the events from EventStore in Log Analytics by configuring your cluster with WAD extension.
- **HealthStore.** Provides a snapshot of the current health of the cluster. A stateful service that aggregates all health data reported by entities in a hierarchy. The data is visualized in [Service Fabric Explorer](#). The HealthStore also monitors application upgrades. You can use health queries in PowerShell, a .NET application, or REST APIs. See, [Introduction to Service Fabric health monitoring](#).
- Consider implementing internal custom watchdog services. Those services can periodically report custom health data such as faulty states of running services. For more information, see [Custom health reports](#). You can read the health reports using the Service Fabric explorer.

## Infrastructure metrics and logs

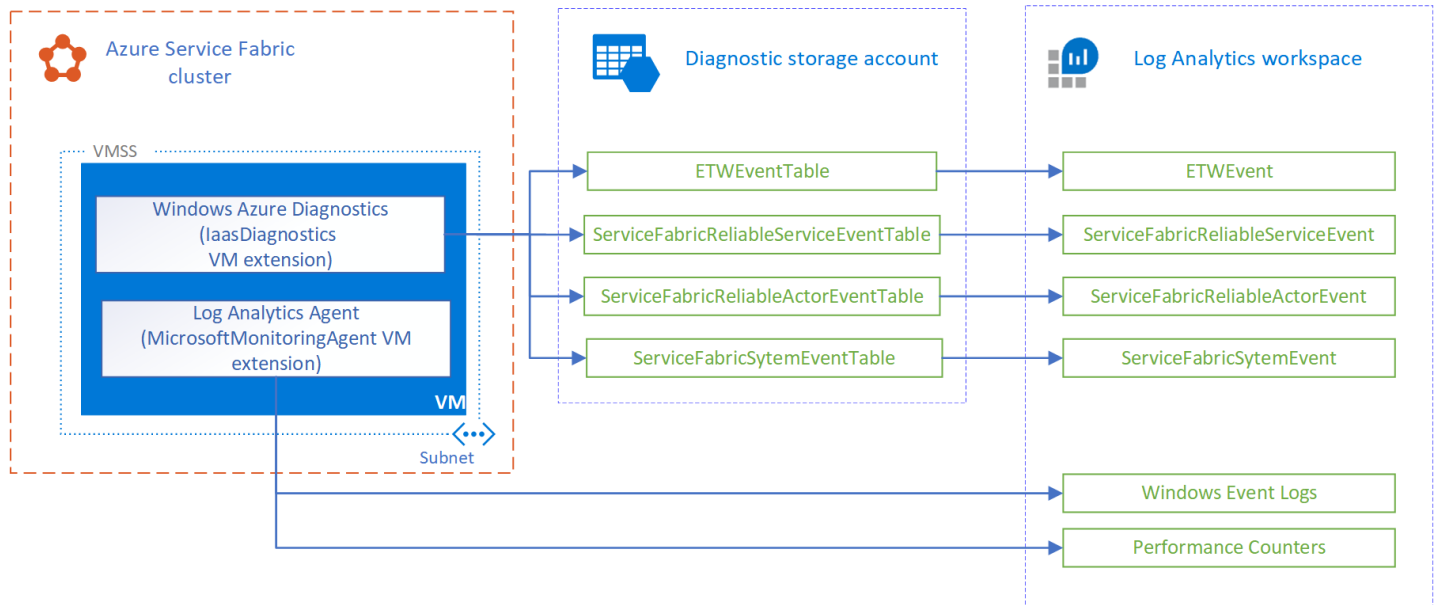
Infrastructure metrics help you to understand resource allocation in the cluster. Here are the main options for collecting this information:

- **Windows Azure Diagnostics (WAD).** Collect logs and metrics at the node level on Windows. You can use WAD by configuring the IaaS.Diagnostics VM extension on any virtual machine scale set that is mapped to a node type to

collect diagnostic events, such as Windows event logs, performance counters, ETW/manifests system and operational events, and custom logs.

- Log Analytics agent. Configure the MicrosoftMonitoringAgent VM extension to send Windows event logs, performance counters, and custom logs to Log Analytics.

There is some overlap in the type of metrics collected through the preceding mechanisms, such as performance counters. Where there is overlap, we recommend using the Log Analytics agent. Because there is no Azure storage for the Log Analytics agent, there is low latency. Also, the performance counters in IaaS Diagnostics cannot be fed into Log Analytics easily.



For information about using VM extensions, see [Azure virtual machine extensions and features](#).

To view the data, configure Log Analytics to view the data collected through WAD. For information about how to configure Log Analytics to read events from a storage account, see [Set up Log Analytics for a cluster](#).

You can also view performance logs and telemetry data related to a Service Fabric cluster, workloads, network traffic, pending updates, and more. See [Performance Monitoring with Log Analytics](#).

[Service Map solution in Log Analytics](#) provides information about the topology of the cluster (that is, the processes running in each node). Send the data in the storage account to [Application Insights](#). There might be some delay in getting data into Application Insights. If you want to see the data real time, consider configuring [Event Hub](#) using sinks and channels. For more information, see [Event aggregation and collection using Windows Azure Diagnostics](#).

## Dependent service metrics

- [Application Insights Application Map](#) provides the topology of the application by using HTTP dependency calls made between services, with the installed Application Insights SDK.
- [Service Map solution in Log Analytics](#) provides information about inbound and outbound traffic from/to external services. In addition, Service Map integrates with other solutions such as updates or security.
- Custom watchdogs can be used to report error conditions on external services. For example, the service could report an error health report if it cannot access an external service or data storage (Azure Cosmos DB).

## Distributed tracing

In microservices architecture, several services often participate to complete a task. The telemetry from each of those services is correlated by using context fields (operation ID, request ID, and so forth) in a distributed trace. By using [Application Map](#) in Application Insights, you can build the view of distributed logical operation and visualize the entire service graph of your application. You can also use transaction diagnostics in Application Insight to correlate server-side telemetry. For more information, see [Unified cross-component transaction diagnostics](#).

[Application Insights Application Map](#) provides the topology of the application by using HTTP dependency calls made between services, with the installed Application Insights SDK. It's also important to correlate tasks that are dispatched asynchronously using a queue. For details about sending correlation telemetry in a queue message, see [Queue instrumentation](#).

For more information, see:

- [Performing a query across multiple resources](#)
- [Telemetry correlation in Application Insights](#)

## Alerts and Dashboards

Application Insights and Log Analytics support an [extensive query language](#) (Kusto query language) that lets you retrieve and analyze log data. Use the queries to create data sets and visualize it in diagnostics dashboards.

Use Azure Monitor alerts to notify sysadmins when certain conditions occur in specific resources. The notification could be an email, Azure function, call a web hook, and so on. For more information, see [Alerts in Azure Monitor](#).

[Log search alert rules](#) allow you to define and run a Kusto query against a Log Analytics workspace at regular intervals. An alert is created if the query result matches a certain condition.

## Next steps

- [Using domain analysis to model microservices](#)
- [Designing a microservices architecture](#)