Contents

[Service bus explorer 1](#_Toc148001795)

[Service Bus 1](#_Toc148001796)

[Topics and subscriptions 2](#_Toc148001797)

[Receive modes 2](#_Toc148001798)

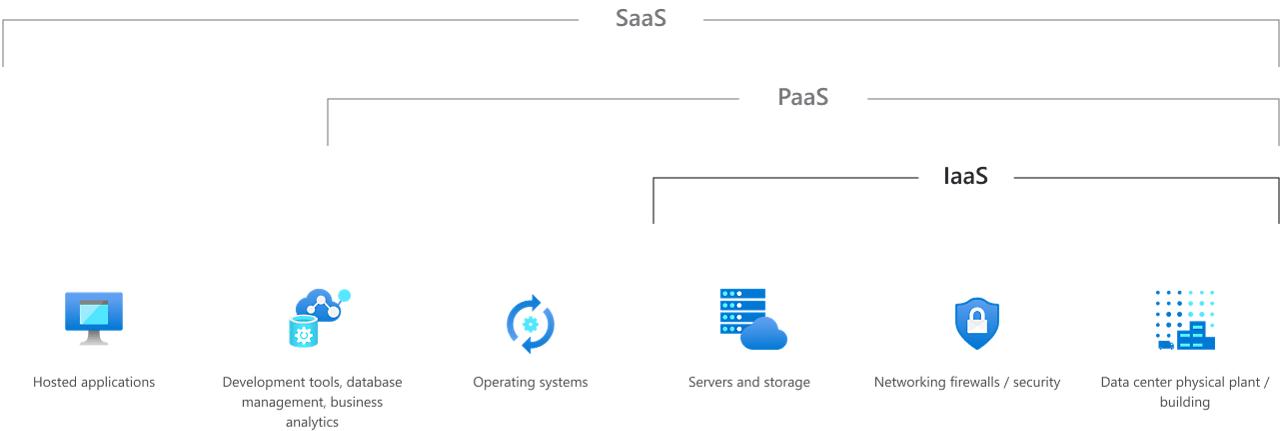
[Filter actions 3](#_Toc148001799)

[Protocols: 3](#_Toc148001800)

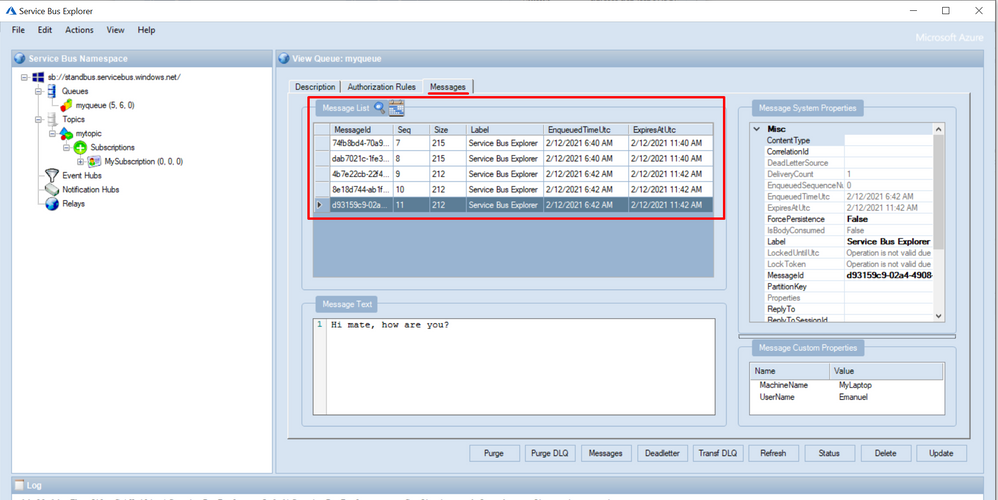
[Optimization 3](#_Toc148001801)

# IAAS PAAS SAAS

Infrastructure as a service (IaaS) is a type of cloud computing service that offers essential compute, storage, and networking resources on demand, on a pay-as-you-go basis. IaaS is one of the four types of cloud services, along with software as a service ([SaaS](https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-saas/)), platform as a service ([PaaS](https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas/)), and [serverless](https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-serverless-computing/).

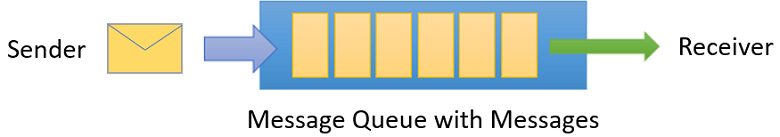


# Service bus explorer



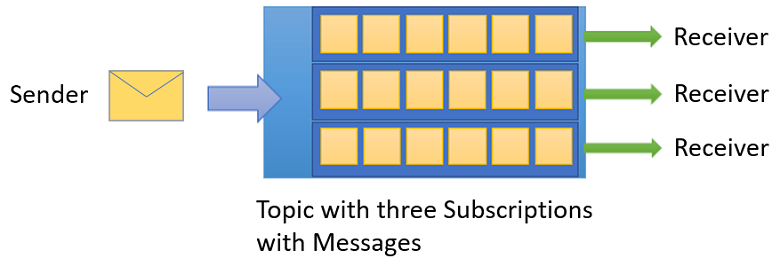
# Service Bus

Queues offer **First In, First Out** (FIFO) message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue. And, only one message consumer receives and processes each message.



# Topics and subscriptions

A queue allows processing of a message by a single consumer. In contrast to queues, topics and subscriptions provide a one-to-many form of communication in a publish and subscribe pattern. It's useful for scaling to large numbers of recipients. Each published message is made available to each subscription registered with the topic. Publisher sends a message to a topic and one or more subscribers receive a copy of the message.



# Receive modes

You can specify two different modes in which consumers can receive messages from Service Bus.

* **Receive and delete**.

when Service Bus receives the request from the consumer, it marks the message as being consumed and returns it to the consumer application. This mode is the simplest model. It works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, consider a scenario in which the consumer issues the receive request and then crashes before processing it. As Service Bus marks the message as consumed, the application begins consuming messages upon restart. It will miss the message that it consumed before the crash. This process is often called **at-most once** processing.

* **Peek lock**. In this mode, the receive operation becomes two-stage, which makes it possible to support applications that can't tolerate missing messages.
  1. Finds the next message to be consumed, **locks** it to prevent other consumers from receiving it, and then, return the message to the application.
  2. After the application finishes processing the message, it requests the Service Bus service to complete the second stage of the receive process. Then, the service **marks the message as consumed**.

If the application is unable to process the message for some reason, it can request the Service Bus service to **abandon** the message. Service Bus **unlocks** the message and makes it available to be received again, either by the same consumer or by another competing consumer. Secondly, there's a **timeout** associated with the lock. If the application fails to process the message before the lock timeout expires, Service Bus unlocks the message and makes it available to be received again.

If the application crashes after it processes the message, but before it requests the Service Bus service to complete the message, Servic

# Filter actions

When a subscription is created, you can supply a filter expression that operates on the properties of the message. The properties can be both the system properties (for example, Label) and custom application properties (for example, StoreName.) The SQL filter expression is optional in this case. Without a SQL filter expression, any filter action defined on a subscription will be done on all the messages for that subscription.

# Protocols:

1. Advanced Message Queuing Protocol (AMQP)
2. Service Bus Messaging Protocol (SBMP)
3. Hypertext Transfer Protocol (HTTP)

# Optimization

1. [Resource planning and considerations](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#resource-planning-and-considerations)
2. [Protocols](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#protocols)
3. [Reusing factories and clients](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#reusing-factories-and-clients)

The Service Bus clients that interact with the service, such as [ServiceBusClient](https://learn.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebusclient), [ServiceBusSender](https://learn.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebussender), [ServiceBusReceiver](https://learn.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebusreceiver), and [ServiceBusProcessor](https://learn.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebusprocessor), should be registered for dependency injection as singletons (or instantiated once and shared). ServiceBusClient can be registered for dependency injection with the [ServiceBusClientBuilderExtensions](https://github.com/Azure/azure-sdk-for-net/blob/master/sdk/servicebus/Azure.Messaging.ServiceBus/src/Compatibility/ServiceBusClientBuilderExtensions.cs).

1. [Concurrent operations](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#concurrent-operations)

To increase the number of operations per time, operations must execute concurrently.

1. [Receive mode](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#receive-mode)

When creating a queue or subscription client, you can specify a receive mode: *Peek-lock* or *Receive and Delete*. The default receive mode is PeekLock. When operating in the default mode, the client sends a request to receive a message from Service Bus. After the client has received the message, it sends a request to complete the message.

When setting the receive mode to ReceiveAndDelete, both steps are combined in a single request. These steps reduce the overall number of operations, and can improve the overall message throughput. This performance gain comes at the risk of losing messages.

1. [Batching store access](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#batching-store-access)

To increase the throughput of a queue, topic, or subscription, Service Bus batches multiple messages when it writes to its internal store.

* With batching on a queue, writing messages, and deleting messages are batched.
* With batching on a topic, writing messages into the store are batched.
* With on a subscription, deleting messages from the store are batched.
* When batched store access is enabled for an entity, Service Bus delays a store write operation for that entity by up to 20 ms.

There is no risk of losing messages with batching, even if there is a Service Bus failure at the end of a 20ms batching interval.

1. [Prefetching](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#prefetching)

Prefetching enables **the queue or subscription client to load additional messages from the service when it receives messages**. The client stores these messages in a local cache. The size of the cache is determined by the QueueClient.PrefetchCount or SubscriptionClient.PrefetchCount properties. Each client that enables prefetching maintains its own cache. A cache isn't shared across clients. If the client starts a receive operation and its cache is empty, the service transmits a batch of messages. The size of the batch equals the size of the cache or 256 KB, whichever is smaller. If the client starts a receive operation and the cache contains a message, the message is taken from the cache.

**When a message is prefetched, the service locks the prefetched message**. With the lock, the prefetched message can't be received by a different receiver. If the receiver can't complete the message before the lock expires, the message becomes available to other receivers. The prefetched copy of the message remains in the cache. The receiver that consumes the expired cached copy receives an exception when it tries to complete that message. **By default, the message lock expires after 60 seconds.** This value can be extended to 5 minutes. To prevent the consumption of expired messages, set the cache size smaller than the number of messages that a client can consume within the lock timeout interval.

When you use the default lock expiration of 60 seconds, a good value for PrefetchCount is 20 times the maximum processing rates of all receivers of the factory.

1. [Prefetching and ReceiveBatch](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#prefetching-and-receivebatch)
2. [Multiple queues or topics](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#multiple-queues-or-topics)
3. [Partitioned namespaces](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#partitioned-namespaces)
4. [Scenarios](https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements?tabs=net-standard-sdk-2#scenarios)

# What are limitations of Azure Functions?

Azure Functions is a serverless compute service provided by Microsoft Azure, and while it offers several benefits, it also has some limitations and constraints that you should be aware of:

1. **Execution Time Limitations:** Azure Functions have execution time limits, which can vary based on the hosting plan. For example, in the Consumption Plan, the maximum execution time is 5 minutes. Long-running processes may not be suitable for Azure Functions without additional strategies like Durable Functions.
2. **Resource Constraints:** Azure Functions run in a constrained environment with limited CPU and memory resources. While this is sufficient for many workloads, resource-intensive tasks may not perform well or may need to be optimized.
3. **Cold Start Latency:** When a function is triggered after a period of inactivity, there can be a noticeable "cold start" delay as the runtime initializes. This can impact the responsiveness of serverless applications.
4. **State Management:** Azure Functions are designed to be stateless, meaning they don't maintain state between invocations by default. Managing and sharing state across function invocations requires additional mechanisms or services.
5. **Limited Execution Environments:** Azure Functions currently support a limited set of programming languages (e.g., C#, JavaScript, Python, PowerShell). If you need a language not supported, you may need to consider other compute options.
6. **Dependency Management:** Managing dependencies and external libraries in Azure Functions can be challenging, especially for complex applications. Dependency management tools and strategies are necessary.
7. **Concurrency and Scaling:** While Azure Functions automatically scale based on incoming requests, there can be limits on concurrent executions, especially in the Consumption Plan. You may need to adjust the hosting plan or use other Azure services for high-concurrency scenarios.
8. **Storage Constraints:** Azure Functions rely on storage for triggers, bindings, and intermediate data. There can be storage-related costs and constraints, and you should design your functions with storage limitations in mind.
9. **Networking Limitations:** Azure Functions have some networking constraints, including limited outbound connections and restrictions on certain network protocols. These limitations can affect communication with external services.
10. **Debugging and Monitoring:** Debugging and monitoring serverless applications, including Azure Functions, can be more challenging compared to traditional applications. Azure provides tools like Application Insights for monitoring and logging.
11. **Limited Execution Environments:** Azure Functions run in a predefined execution environment, which may lack some libraries, binaries, or configurations required for specific tasks. Customizing the execution environment can be challenging.
12. **No Direct Access to Infrastructure:** In serverless environments like Azure Functions, you don't have direct access to the underlying infrastructure, which can limit certain system-level operations and customizations.

Despite these limitations, Azure Functions offer a powerful way to build event-driven, scalable, and cost-effective applications. To mitigate these limitations, it's essential to understand them thoroughly, design your applications accordingly, and consider other Azure services or compute options when necessary.

Top of Form