Autoscaling

05/17/2017 • 13 minutes to read • Contributors all

In this article

Autoscaling is the process of dynamically allocating resources to match performance requirements. As the volume of work grows, an application may need additional resources to maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.

Autoscaling takes advantage of the elasticity of cloud-hosted environments while easing management overhead. It reduces the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.

There are two main ways that an application can scale:

- **Vertical scaling**, also called scaling up and down, means changing the capacity of a resource. For example, you could move an application to a larger VM size. Vertical scaling often requires making the system temporarily unavailable while it is being redeployed. Therefore, it's less common to automate vertical scaling.
- **Horizontal scaling**, also called scaling out and in, means adding or removing instances of a resource. The application continues running without interruption as new resources are provisioned. When the provisioning process is complete, the solution is deployed on these additional resources. If demand drops, the additional resources can be shut down cleanly and deallocated.

Many cloud-based systems, including Microsoft Azure, support automatic horizontal scaling. The rest of this article focuses on horizontal scaling.

> **Note**
>
> Autoscaling mostly applies to compute resources. While it's possible to horizontally scale a database or message queue, this usually involves data partitioning, which is generally not automated.

Overview

An autoscaling strategy typically involves the following pieces:

- Instrumentation and monitoring systems at the application, service, and infrastructure levels. These systems capture key metrics, such as response times, queue lengths, CPU utilization, and memory usage.
- Decision-making logic that evaluates these metrics against predefined thresholds or schedules, and decides whether to scale.
- Components that scale the system.
- Testing, monitoring, and tuning of the autoscaling strategy to ensure that it functions as expected.

Azure provides built-in autoscaling mechanisms that address common scenarios. If a particular service or technology does not have built-in autoscaling functionality, or if you have specific autoscaling requirements beyond its capabilities,

you might consider a custom implementation. A custom implementation would collect operational and system metrics, analyze the metrics, and then scale resources accordingly.

Configure autoscaling for an Azure solution

Azure provides built-in autoscaling for most compute options.

- **Azure Virtual Machines** autoscale via virtual machine scale sets, which manage a set of Azure virtual machines as a group. See How to use automatic scaling and virtual machine scale sets.
- **Service Fabric** also supports autoscaling through virtual machine scale sets. Every node type in a Service Fabric cluster is set up as a separate virtual machine scale set. That way, each node type can be scaled in or out independently. See Scale a Service Fabric cluster in or out using autoscale rules.
- **Azure App Service** has built-in autoscaling. Autoscale settings apply to all of the apps within an App Service. See Scale instance count manually or automatically.
- **Azure Cloud Services** has built-in autoscaling at the role level. See How to configure auto scaling for a Cloud Service in the portal.

These compute options all use Azure Monitor autoscale to provide a common set of autoscaling functionality.

- **Azure Functions** differs from the previous compute options, because you don't need to configure any autoscale rules. Instead, Azure Functions automatically allocates compute power when your code is running, scaling out as necessary to handle load. For more information, see Choose the correct hosting plan for Azure Functions.

Finally, a custom autoscaling solution can sometimes be useful. For example, you could use Azure diagnostics and application-based metrics, along with custom code to monitor and export the application metrics. Then you could define custom rules based on these metrics, and use Resource Manager REST APIs to trigger autoscaling. However, a custom solution is not simple to implement, and should be considered only if none of the previous approaches can fulfill your requirements.

Use the built-in autoscaling features of the platform, if they meet your requirements. If not, carefully consider whether you really need more complex scaling features. Examples of additional requirements may include more granularity of control, different ways to detect trigger events for scaling, scaling across subscriptions, and scaling other types of resources.

Use Azure Monitor autoscale

Azure Monitor autoscale provide a common set of autoscaling functionality for virtual machine scale sets, Azure App Service, and Azure Cloud Service. Scaling can be performed on a schedule, or based on a runtime metric, such as CPU or memory usage. Examples:

- Scale out to 10 instances on weekdays, and scale in to 4 instances on Saturday and Sunday.
- Scale out by one instance if average CPU usage is above 70%, and scale in by one instance if CPU usage falls below 50%.
- Scale out by one instance if the number of messages in a queue exceeds a certain threshold.

For a list of built-in metrics, see Azure Monitor autoscaling common metrics. You can also implement custom metrics by using Application Insights.

You can configure autoscaling by using PowerShell, the Azure CLI, an Azure Resource Manager template, or the Azure portal. For more detailed control, use the Azure Resource Manager REST API. The Azure Monitoring Service Management Library and the Microsoft Insights Library (in preview) are SDKs that allow collecting metrics from different resources, and perform autoscaling by making use of the REST APIs. For resources where Azure Resource

Manager support isn't available, or if you are using Azure Cloud Services, the Service Management REST API can be used for autoscaling. In all other cases, use Azure Resource Manager.

Consider the following points when using Azure autoscale:

- Consider whether you can predict the load on the application accurately enough to use scheduled autoscaling, adding and removing instances to meet anticipated peaks in demand. If this isn't possible, use reactive autoscaling based on runtime metrics, in order to handle unpredictable changes in demand. Typically, you can combine these approaches. For example, create a strategy that adds resources based on a schedule of the times when you know the application is busiest. This helps to ensure that capacity is available when required, without any delay from starting new instances. For each scheduled rule, define metrics that allow reactive autoscaling during that period to ensure that the application can handle sustained but unpredictable peaks in demand.

- It's often difficult to understand the relationship between metrics and capacity requirements, especially when an application is initially deployed. Provision a little extra capacity at the beginning, and then monitor and tune the autoscaling rules to bring the capacity closer to the actual load.

- Configure the autoscaling rules, and then monitor the performance of your application over time. Use the results of this monitoring to adjust the way in which the system scales if necessary. However, keep in mind that autoscaling is not an instantaneous process. It takes time to react to a metric such as average CPU utilization exceeding (or falling below) a specified threshold.

- Autoscaling rules that use a detection mechanism based on a measured trigger attribute (such as CPU usage or queue length) use an aggregated value over time, rather than instantaneous values, to trigger an autoscaling action. By default, the aggregate is an average of the values. This prevents the system from reacting too quickly, or causing rapid oscillation. It also allows time for new instances that are automatically started to settle into running mode, preventing additional autoscaling actions from occurring while the new instances are starting up. For Azure Cloud Services and Azure Virtual Machines, the default period for the aggregation is 45 minutes, so it can take up to this period of time for the metric to trigger autoscaling in response to spikes in demand. You can change the aggregation period by using the SDK, but periods of less than 25 minutes may cause unpredictable results. For Web Apps, the averaging period is much shorter, allowing new instances to be available in about five minutes after a change to the average trigger measure.

- If you configure autoscaling using the SDK rather than the portal, you can specify a more detailed schedule during which the rules are active. You can also create your own metrics and use them with or without any of the existing ones in your autoscaling rules. For example, you may wish to use alternative counters, such as the number of requests per second or the average memory availability, or use custom counters to measure specific business processes.

- When autoscaling Service Fabric, the node types in your cluster are made of virtual machine scale sets at the back end, so you need to set up autoscale rules for each node type. Take into account the number of nodes that you must have before you set up autoscaling. The minimum number of nodes that you must have for the primary node type is driven by the reliability level you have chosen. For more information, see scale a Service Fabric cluster in or out using autoscale rules.

- You can use the portal to link resources such as SQL Database instances and queues to a Cloud Service instance. This allows you to more easily access the separate manual and automatic scaling configuration options for each of the linked resources. For more information, see How to: Link a resource to a cloud service.

- When you configure multiple policies and rules, they could conflict with each other. Autoscale uses the following conflict resolution rules to ensure that there is always a sufficient number of instances running:
  - Scale-out operations always take precedence over scale-in operations.
  - When scale-out operations conflict, the rule that initiates the largest increase in the number of instances takes precedence.
  - When scale in operations conflict, the rule that initiates the smallest decrease in the number of instances takes precedence.

- In an App Service Environment, any worker pool or front-end metrics can be used to define autoscale rules. For more information, see Autoscaling and App Service Environment.

Application design considerations

Autoscaling isn't an instant solution. Simply adding resources to a system or running more instances of a process doesn't guarantee that the performance of the system will improve. Consider the following points when designing an autoscaling strategy:

- The system must be designed to be horizontally scalable. Avoid making assumptions about instance affinity; do not design solutions that require that the code is always running in a specific instance of a process. When scaling a cloud service or web site horizontally, don't assume that a series of requests from the same source will always be routed to the same instance. For the same reason, design services to be stateless to avoid requiring a series of requests from an application to always be routed to the same instance of a service. When designing a service that reads messages from a queue and processes them, don't make any assumptions about which instance of the service handles a specific message. Autoscaling could start additional instances of a service as the queue length grows. The Competing Consumers pattern describes how to handle this scenario.

- If the solution implements a long-running task, design this task to support both scaling out and scaling in. Without due care, such a task could prevent an instance of a process from being shut down cleanly when the system scales in, or it could lose data if the process is forcibly terminated. Ideally, refactor a long-running task and break up the processing that it performs into smaller, discrete chunks. The Pipes and Filters pattern provides an example of how you can achieve this.

- Alternatively, you can implement a checkpoint mechanism that records state information about the task at regular intervals, and save this state in durable storage that can be accessed by any instance of the process running the task. In this way, if the process is shut down, the work that it was performing can be resumed from the last checkpoint by using another instance.

- When background tasks run on separate compute instances, such as in worker roles of a cloud-services–hosted application, you may need to scale different parts of the application using different scaling policies. For example, you may need to deploy additional user interface (UI) compute instances without increasing the number of background compute instances, or the opposite of this. If you offer different levels of service (such as basic and premium service packages), you may need to scale out the compute resources for premium service packages more aggressively than those for basic service packages in order to meet SLAs.

- Consider using the length of the queue over which UI and background compute instances communicate as a criterion for your autoscaling strategy. This is the best indicator of an imbalance or difference between the current load and the processing capacity of the background task.

- If you base your autoscaling strategy on counters that measure business processes, such as the number of orders placed per hour or the average execution time of a complex transaction, ensure that you fully understand the relationship between the results from these types of counters and the actual compute capacity requirements. It may be necessary to scale more than one component or compute unit in response to changes in business process counters.

- To prevent a system from attempting to scale out excessively, and to avoid the costs associated with running many thousands of instances, consider limiting the maximum number of instances that can be automatically added. Most autoscaling mechanisms allow you to specify the minimum and maximum number of instances for a rule. In addition, consider gracefully degrading the functionality that the system provides if the maximum number of instances have been deployed, and the system is still overloaded.

- Keep in mind that autoscaling might not be the most appropriate mechanism to handle a sudden burst in workload. It takes time to provision and start new instances of a service or add resources to a system, and the

peak demand may have passed by the time these additional resources have been made available. In this scenario, it may be better to throttle the service. For more information, see the Throttling pattern.

- Conversely, if you do need the capacity to process all requests when the volume fluctuates rapidly, and cost isn't a major contributing factor, consider using an aggressive autoscaling strategy that starts additional instances more quickly. You can also use a scheduled policy that starts a sufficient number of instances to meet the maximum load before that load is expected.

- The autoscaling mechanism should monitor the autoscaling process, and log the details of each autoscaling event (what triggered it, what resources were added or removed, and when). If you create a custom autoscaling mechanism, ensure that it incorporates this capability. Analyze the information to help measure the effectiveness of the autoscaling strategy, and tune it if necessary. You can tune both in the short term, as the usage patterns become more obvious, and over the long term, as the business expands or the requirements of the application evolve. If an application reaches the upper limit defined for autoscaling, the mechanism might also alert an operator who could manually start additional resources if necessary. Note that under these circumstances the operator may also be responsible for manually removing these resources after the workload eases.

Related patterns and guidance

The following patterns and guidance may also be relevant to your scenario when implementing autoscaling:

- Throttling pattern. This pattern describes how an application can continue to function and meet SLAs when an increase in demand places an extreme load on resources. Throttling can be used with autoscaling to prevent a system from being overwhelmed while the system scales out.

- Competing Consumers pattern. This pattern describes how to implement a pool of service instances that can handle messages from any application instance. Autoscaling can be used to start and stop service instances to match the anticipated workload. This approach enables a system to process multiple messages concurrently to optimize throughput, improve scalability and availability, and balance the workload.

- Monitoring and diagnostics. Instrumentation and telemetry are vital for gathering the information that can drive the autoscaling process.