

Throttling pattern

06/23/2017 • 7 minutes to read • Contributors      all

In this article

[Context and problem](#)

[Solution](#)

[Issues and considerations](#)

[When to use this pattern](#)

[Example](#)

[Related patterns and guidance](#)

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This can allow the system to continue to function and meet service level agreements, even when an increase in demand places an extreme load on resources.

Context and problem

The load on a cloud application typically varies over time based on the number of active users or the types of activities they are performing. For example, more users are likely to be active during business hours, or the system might be required to perform computationally expensive analytics at the end of each month. There might also be sudden and unanticipated bursts in activity. If the processing requirements of the system exceed the capacity of the resources that are available, it'll suffer from poor performance and can even fail. If the system has to meet an agreed level of service, such failure could be unacceptable.

There're many strategies available for handling varying load in the cloud, depending on the business goals for the application. One strategy is to use autoscaling to match the provisioned resources to the user needs at any given time. This has the potential to consistently meet user demand, while optimizing running costs. However, while autoscaling can trigger the provisioning of additional resources, this provisioning isn't immediate. If demand grows quickly, there can be a window of time where there's a resource deficit.

Solution

An alternative strategy to autoscaling is to allow applications to use resources only up to a limit, and then throttle them when this limit is reached. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can throttle requests from one or more users. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place. For more information on monitoring resource usage, see the [Instrumentation and Telemetry Guidance](#).

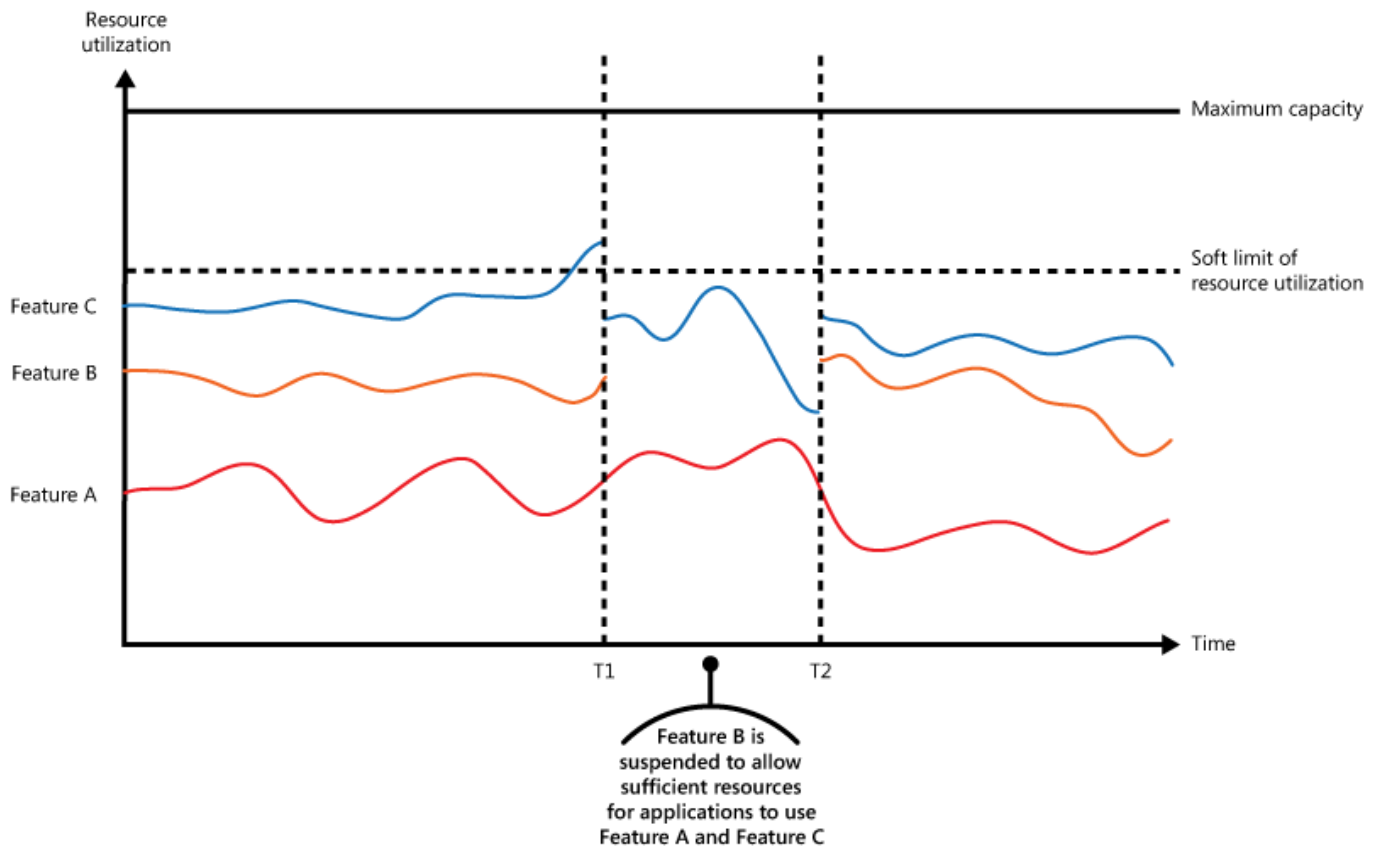
The system could implement several throttling strategies, including:

- Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time. This requires the system to meter the use of resources for each tenant or user running an application. For more information, see the [Service Metering Guidance](#).
- Disabling or degrading the functionality of selected nonessential services so that essential services can run unimpeded with sufficient resources. For example, if the application is streaming video output, it could switch to a lower resolution.
- Using load leveling to smooth the volume of activity (this approach is covered in more detail by the [Queue-based Load Leveling pattern](#)). In a multi-tenant environment, this approach will reduce the performance for every tenant.

If the system must support a mix of tenants with different SLAs, the work for high-value tenants might be performed immediately. Requests for other tenants can be held back, and handled when the backlog has eased. The [Priority Queue pattern](#) could be used to help implement this approach.

- Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.

The figure shows an area graph for resource use (a combination of memory, CPU, bandwidth, and other factors) against time for applications that are making use of three features. A feature is an area of functionality, such as a component that performs a specific set of tasks, a piece of code that performs a complex calculation, or an element that provides a service such as an in-memory cache. These features are labeled A, B, and C.

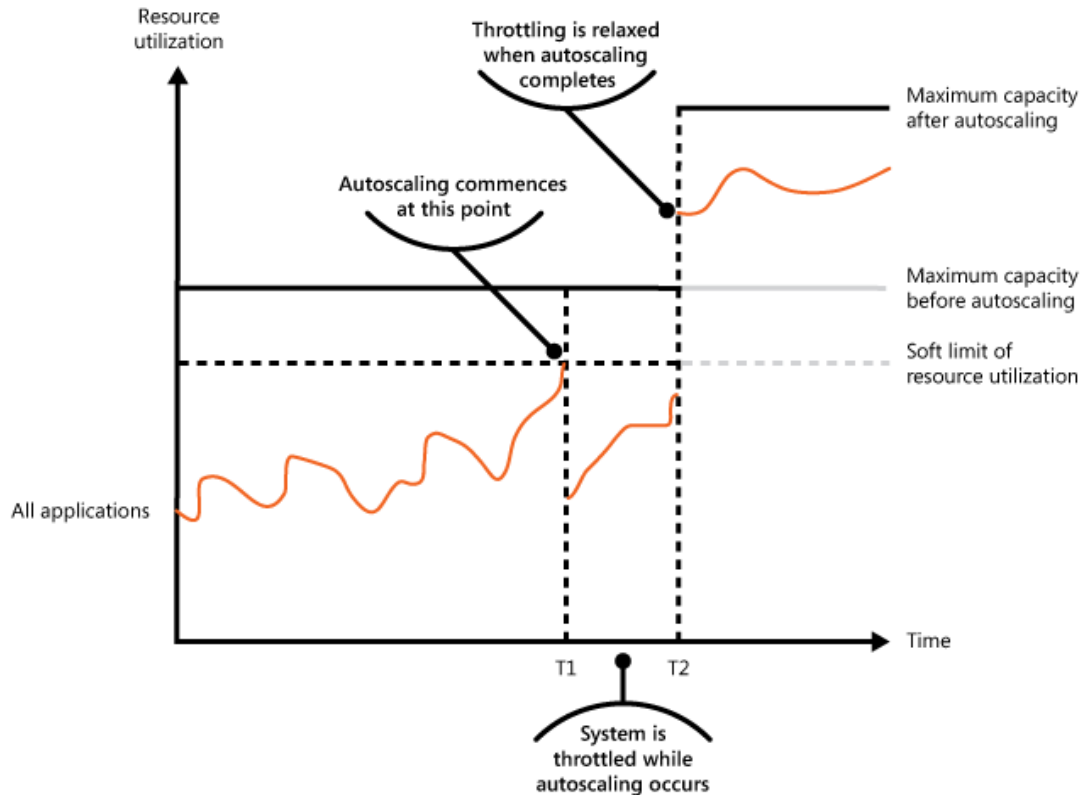


The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature A and Feature B indicates the resources used by applications invoking Feature B. Aggregating the areas for each feature shows the total resource use of the system.

The previous figure illustrates the effects of deferring operations. Just prior to time T1, the total resources allocated to all applications using these features reach a threshold (the limit of resource use). At this point, the applications are in danger of exhausting the resources available. In this system, Feature B is less critical than Feature A or Feature C, so it's temporarily disabled and the resources that it was using are released. Between times T1 and T2, the applications using Feature A and Feature C continue running as normal. Eventually, the resource use of these two features diminishes to the point when, at time T2, there is sufficient capacity to enable Feature B again.

The autoscaling and throttling approaches can also be combined to help keep the applications responsive and within SLAs. If the demand is expected to remain high, throttling provides a temporary solution while the system scales out. At this point, the full functionality of the system can be restored.

The next figure shows an area graph of the overall resource use by all applications running in a system against time, and illustrates how throttling can be combined with autoscaling.



At time T1, the threshold specifying the soft limit of resource use is reached. At this point, the system can start to scale out. However, if the new resources don't become available quickly enough, then the existing resources might be exhausted and the system could fail. To prevent this from occurring, the system is temporarily throttled, as described earlier. When autoscaling has completed and the additional resources are available, throttling can be relaxed.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

- Throttling an application, and the strategy to use, is an architectural decision that impacts the entire design of a system. Throttling should be considered early in the application design process because it isn't easy to add once a system has been implemented.
- Throttling must be performed quickly. The system must be capable of detecting an increase in activity and react accordingly. The system must also be able to revert to its original state quickly after the load has eased. This requires that the appropriate performance data is continually captured and monitored.
- If a service needs to temporarily deny a user request, it should return a specific error code so the client application understands that the reason for the refusal to perform an operation is due to throttling. The client application can wait for a period before retrying the request.
- Throttling can be used as a temporary measure while a system autoscales. In some cases it's better to simply throttle, rather than to scale, if a burst in activity is sudden and isn't expected to be long lived because scaling can add considerably to running costs.
- If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.

When to use this pattern

Use this pattern:

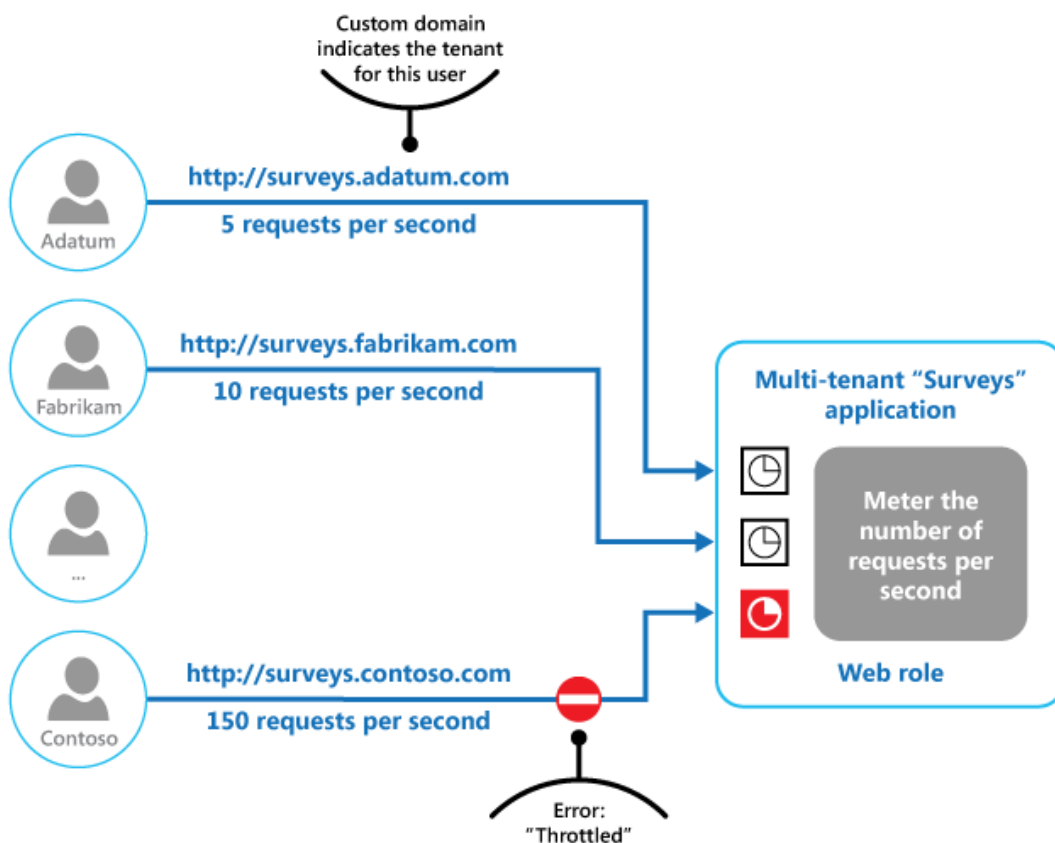
- To ensure that a system continues to meet service level agreements.

- To prevent a single tenant from monopolizing the resources provided by an application.
- To handle bursts in activity.
- To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning.

Example

The final figure illustrates how throttling can be implemented in a multi-tenant system. Users from each of the tenant organizations access a cloud-hosted application where they fill out and submit surveys. The application contains instrumentation that monitors the rate at which these users are submitting requests to the application.

In order to prevent the users from one tenant affecting the responsiveness and availability of the application for all other users, a limit is applied to the number of requests per second the users from any one tenant can submit. The application blocks requests that exceed this limit.



Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Instrumentation and Telemetry Guidance](#). Throttling depends on gathering information about how heavily a service is being used. Describes how to generate and capture custom monitoring information.
- [Service Metering Guidance](#). Describes how to meter the use of services in order to gain an understanding of how they are used. This information can be useful in determining how to throttle a service.
- [Autoscaling Guidance](#). Throttling can be used as an interim measure while a system autoscales, or to remove the need for a system to autoscale. Contains information on autoscaling strategies.
- [Queue-based Load Leveling pattern](#). Queue-based load leveling is a commonly used mechanism for implementing throttling. A queue can act as a buffer that helps to even out the rate at which requests sent by an application are delivered to a service.
- [Priority Queue pattern](#). A system can use priority queuing as part of its throttling strategy to maintain performance for critical or higher value applications, while reducing the performance of less important applications.

