


# Scalability checklist

01/10/2018 • 14 minutes to read • Contributors 

## In this article

[Application design](#)

[Data management](#)

[Implementation](#)

Scalability is the ability of a system to handle increased load, and is one of the [pillars of software quality](#). Use this checklist to review your application architecture from a scalability standpoint.

## Application design

**Partition the workload.** Design parts of the process to be discrete and decomposable. Minimize the size of each part, while following the usual rules for separation of concerns and the single responsibility principle. This allows the component parts to be distributed in a way that maximizes use of each compute unit (such as a role or database server). It also makes it easier to scale the application by adding instances of specific resources. For complex domains, consider adopting a [microservices architecture](#).

**Design for scaling.** Scaling allows applications to react to variable load by increasing and decreasing the number of instances of roles, queues, and other services they use. However, the application must be designed with this in mind. For example, the application and the services it uses must be stateless, to allow requests to be routed to any instance. This also prevents the addition or removal of specific instances from adversely affecting current users. You should also implement configuration or autodetection of instances as they are added and removed, so that code in the application can perform the necessary routing. For example, a web application might use a set of queues in a round-robin approach to route requests to background services running in worker roles. The web application must be able to detect changes in the number of queues, to successfully route requests and balance the load on the application.

**Scale as a unit.** Plan for additional resources to accommodate growth. For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond these limits. Determine the scale units for the system in terms of well-defined sets of resources. This makes applying scale-out operations easier, and less prone to negative impact on the application through limitations imposed by lack of resources in some part of the overall system. For example, adding x number of web and worker roles might require y number of additional queues and z number of storage accounts to handle the additional workload generated by the roles. So a scale unit could consist of x web and worker roles, y queues, and z storage accounts. Design the application so that it's easily scaled by adding one or more scale units.

**Avoid client affinity.** Where possible, ensure that the application does not require affinity. Requests can thus be routed to any instance, and the number of instances is irrelevant. This also avoids the overhead of storing, retrieving, and maintaining state information for each user.

**Take advantage of platform autoscaling features.** Where the hosting platform supports an autoscaling capability, such as Azure Autoscale, prefer it to custom or third-party mechanisms unless the built-in mechanism can't fulfill your requirements. Use scheduled scaling rules where possible to ensure resources are available without a start-up delay, but add reactive autoscaling to the rules where appropriate to cope with unexpected changes in demand. You can use the autoscaling operations in the Service Management API to adjust autoscaling, and to add custom counters to rules. For more information, see [Auto-scaling guidance](#).

**Offload intensive CPU/IO tasks as background tasks.** If a request to a service is expected to take a long time to run or absorb considerable resources, offload the processing for this request to a separate task. Use worker roles or

background jobs (depending on the hosting platform) to execute these tasks. This strategy enables the service to continue receiving further requests and remain responsive. For more information, see [Background jobs guidance](#).

**Distribute the workload for background tasks.** Where there are many background tasks, or the tasks require considerable time or resources, spread the work across multiple compute units (such as worker roles or background jobs). For one possible solution, see the [Competing Consumers pattern](#).

**Consider moving toward a *shared-nothing* architecture.** A shared-nothing architecture uses independent, self-sufficient nodes that have no single point of contention (such as shared services or storage). In theory, such a system can scale almost indefinitely. While a fully shared-nothing approach is generally not practical for most applications, it may provide opportunities to design for better scalability. For example, avoiding the use of server-side session state, client affinity, and data partitioning are good examples of moving toward a shared-nothing architecture.

## Data management

**Use data partitioning.** Divide the data across multiple databases and database servers, or design the application to use data storage services that can provide this partitioning transparently (examples include Azure SQL Database Elastic Database, and Azure Table storage). This approach can help to maximize performance and allow easier scaling. There are different partitioning techniques, such as horizontal, vertical, and functional. You can use a combination of these to achieve maximum benefit from increased query performance, simpler scalability, more flexible management, better availability, and to match the type of store to the data it will hold. Also, consider using different types of data store for different types of data, choosing the types based on how well they are optimized for the specific type of data. This may include using table storage, a document database, or a column-family data store, instead of, or as well as, a relational database. For more information, see [Data partitioning guidance](#).

**Design for eventual consistency.** Eventual consistency improves scalability by reducing or removing the time needed to synchronize related data partitioned across multiple stores. The cost is that data is not always consistent when it is read, and some write operations may cause conflicts. Eventual consistency is ideal for situations where the same data is read frequently but written infrequently. For more information, see the [Data Consistency Primer](#).

**Reduce chatty interactions between components and services.** Avoid designing interactions in which an application is required to make multiple calls to a service (each of which returns a small amount of data), rather than a single call that can return all of the data. Where possible, combine several related operations into a single request when the call is to a service or component that has noticeable latency. This makes it easier to monitor performance and optimize complex operations. For example, use stored procedures in databases to encapsulate complex logic, and reduce the number of round trips and resource locking.

**Use queues to level the load for high velocity data writes.** Surges in demand for a service can overwhelm that service and cause escalating failures. To prevent this, consider implementing the [Queue-Based Load Leveling pattern](#). Use a queue that acts as a buffer between a task and a service that it invokes. This can smooth intermittent heavy loads that may otherwise cause the service to fail or the task to time out.

**Minimize the load on the data store.** The data store is commonly a processing bottleneck, a costly resource, and often not easy to scale out. Where possible, remove logic (such as processing XML documents or JSON objects) from the data store, and perform processing within the application. For example, instead of passing XML to the database (other than as an opaque string for storage), serialize or deserialize the XML within the application layer and pass it in a form that is native to the data store. It's typically much easier to scale out the application than the data store, so you should attempt to do as much of the compute-intensive processing as possible within the application.

**Minimize the volume of data retrieved.** Retrieve only the data you require by specifying columns and using criteria to select rows. Make use of table value parameters and the appropriate isolation level. Use mechanisms like entity tags to avoid retrieving data unnecessarily.

**Aggressively use caching.** Use caching wherever possible to reduce the load on resources and services that generate or deliver data. Caching is typically suited to data that is relatively static, or that requires considerable processing to

obtain. Caching should occur at all levels where appropriate in each layer of the application, including data access and user interface generation. For more information, see the [Caching Guidance](#).

**Handle data growth and retention.** The amount of data stored by an application grows over time. This growth increases storage costs as well as latency when accessing the data, affecting application throughput and performance. It may be possible to periodically archive some of the old data that is no longer accessed, or move data that is rarely accessed into long-term storage that is more cost efficient, even if the access latency is higher.

**Optimize Data Transfer Objects (DTOs) using an efficient binary format.** DTOs are passed between the layers of an application many times. Minimizing the size reduces the load on resources and the network. However, balance the savings with the overhead of converting the data to the required format in each location where it is used. Adopt a format that has the maximum interoperability to enable easy reuse of a component.

**Set cache control.** Design and configure the application to use output caching or fragment caching where possible, to minimize processing load.

**Enable client side caching.** Web applications should enable cache settings on the content that can be cached. This is commonly disabled by default. Configure the server to deliver the appropriate cache control headers to enable caching of content on proxy servers and clients.

**Use Azure blob storage and the Azure Content Delivery Network to reduce the load on the application.** Consider storing static or relatively static public content, such as images, resources, scripts, and style sheets, in blob storage. This approach relieves the application of the load caused by dynamically generating this content for each request. Additionally, consider using the Content Delivery Network to cache this content and deliver it to clients. Using the Content Delivery Network can improve performance at the client because the content is delivered from the geographically closest datacenter that contains a Content Delivery Network cache. For more information, see the [Content Delivery Network Guidance](#).

**Optimize and tune SQL queries and indexes.** Some T-SQL statements or constructs may have an adverse effect on performance that can be reduced by optimizing the code in a stored procedure. For example, avoid converting **datetime** types to a **varchar** before comparing with a **datetime** literal value. Use date/time comparison functions instead. Lack of appropriate indexes can also slow query execution. If you use an object/relational mapping framework, understand how it works and how it may affect performance of the data access layer. For more information, see [Query Tuning](#).

**Consider denormalizing data.** Data normalization helps to avoid duplication and inconsistency. However, maintaining multiple indexes, checking for referential integrity, performing multiple accesses to small chunks of data, and joining tables to reassemble the data imposes an overhead that can affect performance. Consider if some additional storage volume and duplication is acceptable in order to reduce the load on the data store. Also consider if the application itself (which is typically easier to scale) can be relied on to take over tasks such as managing referential integrity in order to reduce the load on the data store. For more information, see [Data partitioning guidance](#).

## Implementation

**Review the performance antipatterns.** See [Performance antipatterns for cloud applications](#) for common practices that are likely to cause scalability problems when an application is under pressure.

**Use asynchronous calls.** Use asynchronous code wherever possible when accessing resources or services that may be limited by I/O or network bandwidth, or that have a noticeable latency, in order to avoid locking the calling thread.

**Avoid locking resources, and use an optimistic approach instead.** Never lock access to resources such as storage or other services that have noticeable latency, because this is a primary cause of poor performance. Always use optimistic approaches to managing concurrent operations, such as writing to storage. Use features of the storage layer to manage conflicts. In distributed applications, data may be only eventually consistent.

**Compress highly compressible data over high latency, low bandwidth networks.** In the majority of cases in a web application, the largest volume of data generated by the application and passed over the network is HTTP responses to client requests. HTTP compression can reduce this considerably, especially for static content. This can reduce cost as well as reducing the load on the network, though compressing dynamic content does apply a fractionally higher load on the server. In other, more generalized environments, data compression can reduce the volume of data transmitted and minimize transfer time and costs, but the compression and decompression processes incur overhead. As such, compression should only be used when there is a demonstrable gain in performance. Other serialization methods, such as JSON or binary encodings, may reduce the payload size while having less impact on performance, whereas XML is likely to increase it.

**Minimize the time that connections and resources are in use.** Maintain connections and resources only for as long as you need to use them. For example, open connections as late as possible, and allow them to be returned to the connection pool as soon as possible. Acquire resources as late as possible, and dispose of them as soon as possible.

**Minimize the number of connections required.** Service connections absorb resources. Limit the number that are required and ensure that existing connections are reused whenever possible. For example, after performing authentication, use impersonation where appropriate to run code as a specific identity. This can help to make best use of the connection pool by reusing connections.

🚨 **Note**

APIs for some services automatically reuse connections, provided service-specific guidelines are followed. It's important that you understand the conditions that enable connection reuse for each service that your application uses.

**Send requests in batches to optimize network use.** For example, send and read messages in batches when accessing a queue, and perform multiple reads or writes as a batch when accessing storage or a cache. This can help to maximize efficiency of the services and data stores by reducing the number of calls across the network.

**Avoid a requirement to store server-side session state** where possible. Server-side session state management typically requires client affinity (that is, routing each request to the same server instance), which affects the ability of the system to scale. Ideally, you should design clients to be stateless with respect to the servers that they use. However, if the application must maintain session state, store sensitive data or large volumes of per-client data in a distributed server-side cache that all instances of the application can access.

**Optimize table storage schemas.** When using table stores that require the table and column names to be passed and processed with every query, such as Azure table storage, consider using shorter names to reduce this overhead. However, do not sacrifice readability or manageability by using overly compact names.

**Create resource dependencies during deployment or at application startup.** Avoid repeated calls to methods that test the existence of a resource and then create the resource if it does not exist. Methods such as *CloudTable.CreateIfNotExists* and *CloudQueue.CreateIfNotExists* in the Azure Storage Client Library follow this pattern. These methods can impose considerable overhead if they are invoked before each access to a storage table or storage queue. Instead:

- Create the required resources when the application is deployed, or when it first starts (a single call to *CreateIfNotExists* for each resource in the startup code for a web or worker role is acceptable). However, be sure to handle exceptions that may arise if your code attempts to access a resource that doesn't exist. In these situations, you should log the exception, and possibly alert an operator that a resource is missing.
- Under some circumstances, it may be appropriate to create the missing resource as part of the exception handling code. But you should adopt this approach with caution as the non-existence of the resource might be indicative of a programming error (a misspelled resource name for example), or some other infrastructure-level issue.

**Use lightweight frameworks.** Carefully choose the APIs and frameworks you use to minimize resource usage, execution time, and overall load on the application. For example, using Web API to handle service requests can reduce the application footprint and increase execution speed, but it may not be suitable for advanced scenarios where the additional capabilities of Windows Communication Foundation are required.

**Consider minimizing the number of service accounts.** For example, use a specific account to access resources or services that impose a limit on connections, or perform better where fewer connections are maintained. This approach is common for services such as databases, but it can affect the ability to accurately audit operations due to the impersonation of the original user.

**Carry out performance profiling and load testing** during development, as part of test routines, and before final release to ensure the application performs and scales as required. This testing should occur on the same type of hardware as the production platform, and with the same types and quantities of data and user load as it will encounter in production. For more information, see [Testing the performance of a cloud service](#).